IMPLEMENTATION AND ANALYSIS OF NP-COMPLETE ALGORITHMS ON A DISTRIBUTED MEMORY COMPUTER

THESIS

Joel Shane Garmon
Captain, USAF

AFIT/GE/ENG/92-M

AFIT/GE/ENG/92-M-01

IMPLEMENTATION AND ANALYSIS OF NP-COMPLETE ALGORITHMS ON A DISTRIBUTED MEMORY COMPUTER

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

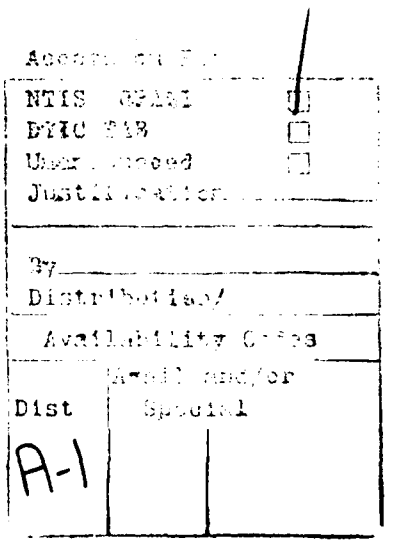
Requirements for the Degree of

Master of Science in Electrical Engineering

Joel Shane Garmon,

Captain, USAF

March, 1992



Approved for public release; distribution unlimited

Table of Contents

		Page
Table of Contents		ii
List of Figures		vi
List of Tables		ix
Abstract		x
I.	Introduction	1-1
	1.1 General Problem Statement	1-1
	1.2 Background	1-2
	1.3 Scope	1-4
	1.4 Summary of the Thesis	1-6
II.	Background and Requirements	2-1
	2.1 Introduction	2-1
	2.2 NP-Complete	2-1
	2.3 Parallel Architectures	2-4
	2.3.1 Hypercube Architecture	2-6
	2.3.2 Granularity	2-6
	2.4 Parallel Search Issues	2-9
	2.4.1 Global Variable Communication	2-9
	2.4.2 Task Allocation Methods	2-10
	2.4.3 Limits on Speedup and Efficiency	2-11
	2.5 General Search Techniques	2-12
	2.5.1 Greedy Method	2-12
	2.5.2 Uninformed Search	2-12

Page

2.5.3 Best First Search 2-16

2.6 Summary of Search Algorithms 2-21

III. Methodology and Design 3-1

3.1 Introduction 3-1

3.2 Methodology 3-1

3.3 Metrics 3-2

3.3.1 Speedup 3-2

3.3.2 Nodes Expanded 3-3

3.4 Understanding the Problem 3-9

3.4.1 Traveling Salesman Problem 3-9

3.4.2 A* 3-10

3.5 Heuristic Estimate of $h(n)$ 3-11

3.5.1 Assignment Problem Example 3-13

3.5.2 Assignment Problem Algorithm 3-13

3.6 High Level Design 3-21

3.6.1 Sequential TSP Algorithm 3-21

3.6.2 Decomposition Techniques 3-25

3.6.3 High Level Algorithms 3-26

3.7 Summary 3-28

IV. Low Level Design and Implementation 4-1

4.1 Introduction 4-1

4.2 Data Structures 4-1

4.3 Low Level Design 4-7

4.3.1 Random City Generator 4-7

4.3.2 Control Program 4-7

4.3.3 Worker Program 4-9

Page

4.3.4 Host Program 4-10

4.3.5 Subroutines 4-11

4.4 Distributed List 4-15

4.4.1 DL Without Load Balancing 4-15

4.4.2 DL With Load Balancing 4-16

4.5 A* Variations 4-21

4.5.1 IDA* 4-22

4.5.2 TSP with Levels 4-23

4.5.3 Distributed List with Load Balancing and NODE Distribution 4-24

4.6 Summary 4-31

V. Results 5-1

5.0.1 Introduction 5-1

5.1 Metrics 5-1

5.2 Testing 5-3

5.3 Test Results 5-6

5.4 Summary 5-8

VI. Conclusions and Recommendations for Further Work 6-1

6.1 Introduction 6-1

6.2 Interpretation of the Results 6-2

6.2.1 Preliminary Depth First Search (DFS) 6-3

6.2.2 Evaluation of the Algorithms 6-3

6.2.3 Small Scale Parallel Computers 6-4

6.2.4 Large Scale Parallel Computers 6-8

6.2.5 Comparison of DL_LB and DL_DIST Algorithms 6-10

6.3 IDA* Versus Centralized List 6-16

6.4 Guidelines for Distributed Memory Computer Implementation of A* Algorithms 6-17

Page

6.5 Recommendation for Further Research 6-20

6.6 Summary 6-21

Appendix A. Structure Charts A-1

A.1 Introduction A-1

A.2 Centralized List Algorithm A-1

A.3 Distributed List Algorithms A-3

A.3.1 Distributed List with Load Balancing A-3

A.3.2 Distributed List with Load Balancing and Distribution A-4

Appendix B. Test Results and Data B-1

B.1 Introduction B-1

B.2 Data B-1

B.2.1 Execution Time Graphs B-7

B.2.2 States Expanded Graph B-10

B.2.3 Share Data B-12

B.2.4 Distribution Data B-18

B.3 IDA* Data B-24

Appendix C. Problem Definition and Data C-1

C.1 Introduction C-1

C.2 Problem n22a C-1

C.3 Problem n55a C-2

C.4 Problem n65a C-6

C.5 Problem n100a C-12

List of Figures

Figure		Page
2.1.	Space Time Relationships	2-2
2.2.	Hypercube Dimensions	2-7
2.3.	Depth First Search	2-13
2.4.	Breadth First Search	2-17
2.5.	Hierarchical Diagram of Best First Algorithms	2-17
2.6.	Iterative Deepening A*	2-20
3.1.	Total Search Space	3-4
3.2.	Search Space For Depth First Search	3-5
3.3.	Search Space For Breadth First Search	3-7
3.4.	Search Space For Best First Search	3-8
3.5.	Assignment Problem Cost Matrix	3-14
3.6.	Assignment Problem Part 1	3-17
3.7.	Assignment Problem Part 2	3-18
3.8.	Assignment Problem Part 3	3-19
3.9.	Assignment Problem Part 4	3-20
3.10.	Example Search Graph for 5 Cities	3-24
4.1.	Structure of Type NODE	4-1
4.2.	Structure of the OPEN Queue	4-3
4.3.	Inserting into the OPEN Queue	4-4
4.4.	Deleting from the OPEN Queue	4-5
4.5.	Cost Matrix Example	4-6
4.6.	Initial OPEN Lists	4-25
4.7.	After Node 0 Distributed	4-26
4.8.	After Node 1 Distributed	4-27

Figure Page

4.9. After Node 2 Distributed 4-28

5.1. TSP for 4 Cities using File n4a 5-4

5.2. Search Graph for 4 Cities using File n4a 5-5

A.1. Centralized List Host Structure Chart A-1

A.2. Control Structure Chart A-2

A.3. Centralized List Worker Structure Chart A-2

A.4. Distributed List Host Structure Chart A-3

A.5. Distributed List Worker Structure Chart A-3

A.6. Distributed List Host Structure Chart A-4

A.7. Distributed List Worker Structure Chart A-4

B.1. Execution Time Data for 22 Cities B-7

B.2. Execution Time Data for 55 Cities B-8

B.3. Execution Time Data for 65 Cities B-8

B.4. Execution Time Data for 100 Cities B-9

B.5. States Expanded Data for 22 Cities B-10

B.6. States Expanded Data for 55 Cities B-10

B.7. States Expanded Data for 65 Cities B-11

B.8. States Expanded Data for 100 Cities B-11

B.9. Execution Timefor 22 Cities B-12

B.10. Execution Time for 55 Cities B-12

B.11. Execution Time for 65 and 100 Cities B-15

B.12. States Expanded for 22 Cities B-15

B.13. States Expanded for 55 Cities B-16

B.14. States Expanded for 65 Cities B-16

B.15. States Expanded for 100 Cities B-17

B.16. Execution Time for 22 Cities B-18

Figure Page

B.17.Execution Time for 55 Cities B-18

B.18.Execution Time for 65 Cities B-21

B.19.Execution Time for 100 Cities B-21

B.20.States Expanded for 22 Cities B-22

B.21.States Expanded for 55 Cities B-22

B.22.States Expanded for 65 Cities B-23

B.23.States Expanded for 100 Cities B-23

B.24.IDA* Execution Time Data B-24

B.25.IDA* States Expanded Data B-27

B.26.Level Execution Time Data B-27

B.27.Level States Expanded Data B-28

List of Tables

Table		Page
2.1.	Memory and Time Comparisons of Search Algorithms	2-22
2.2.	Applications and Implementations of Search Algorithms	2-22
6.1.	States expanded by processor using CL and 100 cities	6-19
B.1.	Centralized List Data	B-2
B.2.	Distributed List with no Load Balancing Data	B-3
B.3.	Distributed List with Load Balancing Data	B-4
B.4.	Distributed List with Load Balancing and Distribution 1 of 2	B-5
B.5.	Distributed List with Load Balancing and Distribution 2 of 2	B-6
B.6.	Share Data 1 of 2	B-13
B.7.	Share Data 2 of 2	B-14
B.8.	Distribution Data 1 of 2	B-19
B.9.	Distribution Data 2 of 2	B-20
B.10.	IDA* Data	B-25
B.11.	Centralized List using Levels Data	B-26

AFIT/GE/ENG/92-M

Abstract

The purpose of this research is to explore methods used to parallelize NP-complete problems and the degree of improvement that can be realized using different methods of load balancing.

A serial and four parallel A* branch and bound algorithms were implemented and executed on an Intel iPSC/2 hypercube computer. One parallel algorithm used a global, or centralized, list to store unfinished work and the other three parallel algorithms used a distributed list to store unfinished work locally on each processor.

The three distributed list algorithms are: without load balancing, with load balancing, and with load balancing and work distribution. The difference between load balancing and work distribution is load balancing only occurs when a processor becomes idle and work distribution attempts to emulate the global list of unfinished work by sharing work throughout the algorithm, not just at the end. Factors which effect when and how often to load balance are also investigated.

Which algorithm performed best depended on how many processors were used to solve the problem. For a small number of processors, 16 or less, the centralized list algorithm easily outperformed all others. However, after 16 processors, the overhead of all processors trying to communicate and request work from the same centralized list began to outweigh any benefits of having a global list. Now the distributed list algorithms began to perform best. When using 32 processors, the distributed list with load balancing and work distribution out performed the other algorithms.

IMPLEMENTATION AND ANALYSIS OF NP-COMPLETE ALGORITHMS ON A DISTRIBUTED MEMORY COMPUTER

I. Introduction

1.1 General Problem Statement

The Department of Defense today is tasked with performing the same mission as ten years ago, but with fewer personnel and less equipment. Sophisticated technology controlled by computers allows the United States to continue its military leadership of the world. To continue this leadership, algorithms must become more efficient as the tasks required of them become more complex.

One of the most widely used problem solving techniques is exhaustive search, which searches all possible answers and selects the best solution. But what happens if the answer to the overall problem depends on the answer to many sub-problems within the main problem? Every possible combination of answers must be investigated to find the best or optimal solution. Combinatorial searches for small problems are possible, but the number of possible solutions which must be checked can expand exponentially beyond our limits in time and memory space to search them. Many real world problems in artificial intelligence, operations research, VLSI chip layout and wire routing, and weapon to target assignment problems can use this exhaustive search technique.

Combinatorial searches whose execution times increase exponentially with a linear addition of information to the problem are in the class of non-deterministic polynomial (NP) complete problems. Examples of NP-complete problems include the knapsack problem, the traveling salesman problem, the set covering problem, the assignment problem, and many others. This research investigates elementary heuristics used to solve NP-complete problems on distributed memory computers.

the search space, the example of 2 weapons and 10 targets requires 1024 bytes. However, when their are 50 targets, the memory requirements grow to 1,130,000,000,000,000 bytes or 1,130 giga bytes. [Carpenter, 1986: 35] This amount of memory is available on few computers. Obviously, more efficient methods must be found for solving these problems. However, as is shown in Chapter II, NP-complete problems can require polynomial space if properly designed.

One method to shorten the time to find a solution is to accept a less than optimal solution. Pearl provides heuristics, or guidelines, using an error function as a bound on the solution. This allows any solution within a predetermined range to be accepted as a solution . Probabilistic methods such as Monte Carlo algorithms control the search based on probabilities of finding a solution down a certain path. This method can also return a less than optimal solution [Pearl,1984: 86-89]. Another non-optimal search method is the *genetic* algorithm. This algorithm solves problems by manipulating strings of instructions the same way chromosomes manipulate DNA. The process involves a complex search that combines blind groping with precise accounting [Antonoff, 1991:70] Since this research only considers optimal solutions, none of these techniques are investigated.

Another method to shorten the time to find a solution is to increase the computing power of the computer. In the past, programmers used faster computers to solve NP-complete problems as the problems increased in complexity. As Hennessy and Joupppi point out, the two most important factors in the high growth rate in computing power is the dramatic increase in the number of transistors available on chip and architectural advances including the use of RISC ideas, pipelining, and caches. With all these improvements, central processor unit (CPU) performance has increased 95,000% since 1980. [Hennessy and Jouppi, 1991: 19-23].

However, traditional sequential computers are approaching the theoretical limit of the time required to perform a computation. The limiting factor on computational speed is propagation delay of signals between transistors on the same chip. This propagation delay consists of a gate delay caused by the transistor itself and signal travel time between transistors. Gate delay has been

reduced to the point where signal travel time between transistors is the dominant delay. This travel delay is being reduced by making the transistors smaller and placing the transistors closer together on the chip. While the number of transistors on a chip can be quite high, there is obviously a limit to the size and space required for each transistor. Since the speed of light limits the time required for a signal to travel between transistors, other methods to improve computation power are being sought [DeCegama, 1989: 23-27].

The design time for main frame and minicomputers is approximately four to five years while that of a microcomputer is approximately two years. According to Hennessy and Jouppi, this shorter design time allows the computers based on microprocessor technology to take full advantage of the rapid changes in VLSI technology and changes in computer architecture. They show that the computing power and speed of microcomputers is on par with mainframes and is quickly approaching that of uniprocessor supercomputers. [Hennessy and Jouppi, 1991:19]. Bell contends computers built using microprocessors connected together to form a multiprocessor computer is the trend of the future in computing [Bell, 1989: 1093-1097].

To obtain more computations in the same amount of time, multiprocessor computers are now used. Each task is divided into sub-tasks and assigned to one of many processors in the computer. By using multiprocessors, a solution to the task may be found in less time than required for the sequential computer.

1.3 Scope

This research investigates the use of multiprocessor computers to solve NP-complete problems. To do this, search algorithms are designed and implemented on an Intel iPSC/2 hypercube. Various search strategies such as depth first search, breadth first search, backtracking, best first search, and branch and bound are incorporated into different algorithms to determine their effect on the algorithm's efficiency. Since this research is concerned only with optimal solutions, non-optimal

algorithms such as probabilistic or genetic algorithms are not considered. User applications to test the search algorithm are also designed and implemented on the hypercube.

The goal of this research is to address the following:

1. Study and analyze previous works
2. Investigate the effects of different static and dynamic load balancing techniques.
3. Investigate when and how to communicate global information.
4. Investigate the effects of keeping the list of work to be done in a centralized list on one master processor or on distributed lists on many processors.
5. Determine the type of algorithm, or combination of algorithms, which best suit a particular problem.
6. Develop appropriate performance metrics to evaluate each algorithm.
7. Investigate the amount of communication vs computation in each algorithm.
8. Investigate the underlying heuristics common to all NP-complete problems.

While this list is far from complete, the time constraint placed upon this research limits the topics which can be investigated.

Many different metrics are used to evaluate the time efficiency of a parallel search algorithm. This thesis investigation bases performance of an algorithm primarily on speedup and number of states generated by the algorithm. Other metrics considered include processor idle time, efficiency, and the ratio of communication versus computation time. Metrics to measure the space efficiency are not considered due to the limited time for this research.

A comprehensive literature search covering the topics of search algorithms, parallel processing, performance analysis, and hypercube computers provides the foundation for this research and is

discussed in chapter 2. Analyzing search algorithms developed at AFIT or stored at software repositories around the country provides additional understanding of the problem.

Since parallel algorithms are especially difficult to design and implement, this research follows standard software engineering practices for documenting, testing, and designing programs.

Since the sequential algorithm is the standard against which the parallel algorithms are initially measured, the first task in the research effort is designing and implementing a sequential A* algorithm. After determining critical parameters of the program, they are measured which provides a baseline for comparison to later versions of the program. After implementing the sequential algorithm on the parallel computer, baseline measurements of critical parameters are again taken. Changes to the parallel program are made and the parameters again measured. After each change to the program, data was collected and analyzed to determine the effect of the changes and to help determine what change to make next. All data was analyzed looking for fundamental heuristics to solving NP complete problems on parallel computers.

1.4 Summary of the Thesis

In this chapter, a working definition of NP-complete problems is provided and a quick example to justify the need to study and improve the methods for solving them on a distributed memory computer. The scope of the research is then presented.

The rest of the thesis is composed of five additional chapters. Chapter II is the literature search to determine the current state of the art. Chapter III provides the high level design of the algorithms and the measurement criteria. The low level design is provided in Chapter IV. Chapter V discusses the results received from the different algorithms and Chapter VI presents my conclusions and recommendations for future work.

This thesis assumes a general understanding of sequential and parallel computers along with some understanding of the search techniques. A quick explanation of concepts and ideas is provided in the following chapters and references are given for further study.

II. Background and Requirements

2.1 Introduction

To perform this research an understanding of NP-complete problems, search techniques, and parallel architectures is essential. Each of these topics is discussed in its own main section. Also, a section discussing current topics in parallel search techniques is provided. Each of these subjects has been extensively studied in books and journals, so only an overview of the subjects is provided here. References are listed to provide a more in-depth study of each topic if desired.

2.2 NP-Complete

Brassard and Brantley define NP-complete problems in terms of two conditions. The first condition is that the problem be a member of NP space. A problem is in NP space if it can be solved on a non-deterministic Turing machine (NDTM) in polynomial time. Since a NDTM is a computing model which can solve an infinite number of problems in parallel, it can solve both polynomial and nonpolynomial time problems in polynomial time.

The second condition required for a problem to be NP-complete is that NP-complete problems must be transformable to every other NP-complete problem in polynomial time [Brassard and Brantley, 1985: 323-325]. Therefore, if a polynomial time solution is found to any of the NP-complete problems, all can be solved in polynomial time. One goal of this research is to investigate the heuristics which are common to parallel NP-complete problems.

Aho and others provide relationships between different classes of problems as shown in Figure 2.1. They also prove that P-space is identically equal to NP-space. Therefore, if a problem is in NP-time, it is in P-space [Aho and others, 1974: 395]. This figure also shows the possibility that other problems in NP-time are also NP-complete.

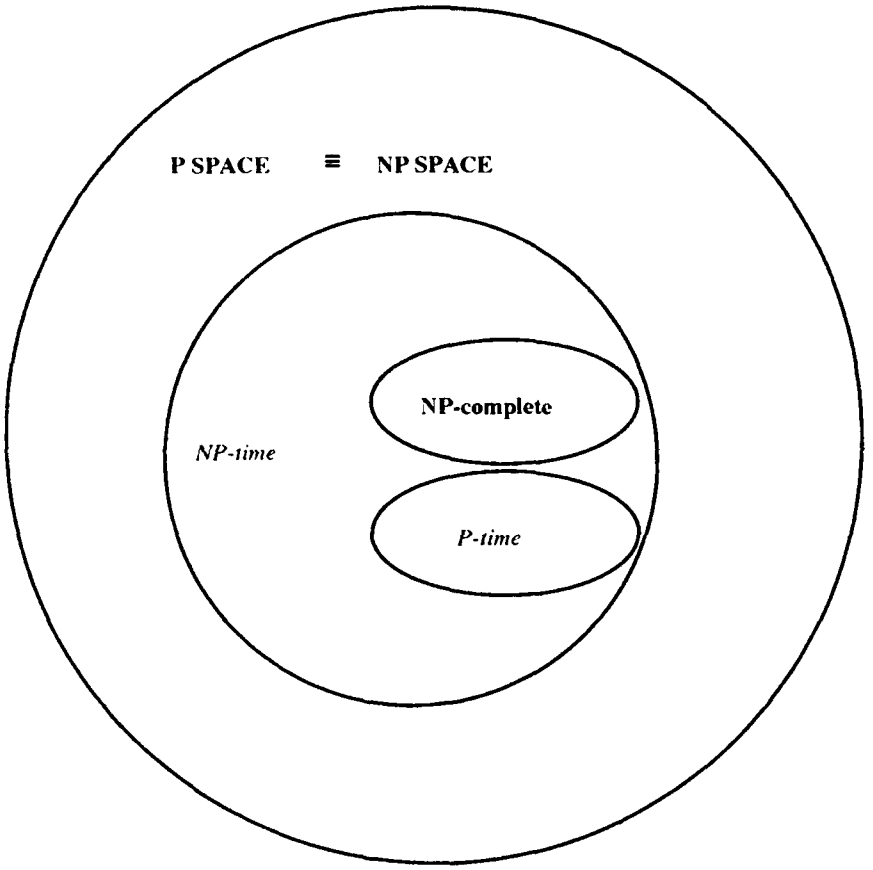


Figure 2.1. Space Time Relationships

Aho lists the following as NP-complete problems:

1. Satisfiability -- Is a Boolean expression satisfiable?
2. Clique — Does an undirected graph have a clique of size k?
3. Hamilton Circuit — Does an undirected graph have a Hamilton circuit?
4. Colorability — Is an undirected graph k-colorable?
5. Feedback Vertex Set -- Does a directed graph have a feed back vertex set with k members?
6. Feedback Edge Set -- Does a directed graph have a feedback edge set with k members?
7. Directed Hamilton Circuit -- Does a directed graph have a directed Hamilton circuit?
8. Set Cover -- Given a family of sets $S_1, S_2, \ldots, S_n$ does there exist a subfamily of k sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ such that
$$\bigcup_{j=1}^{k} S_{i_j} = \bigcup_{j=1}^{n} S_j$$
9. Exact Cover — Given a family of sets $S_1, S_2, \ldots, S_n$ does there exist a set cover consisting of a subfamily of pairwise disjoint sets?

[Aho and others, 1974: 379]. See Aho or Christifides for a more detailed explanation of the above listed problems [Christifides, 1974: 1-75].

The working definition used in this research is the class of problems for which the time complexity has an exponential function as a lower bound. The Traveling Salesman Problem (TSP), the Set Covering Problem (SCP), the assignment problem, and the Knapsack Problem are all a subset of one of the above mentioned NP-complete problems and are themselves NP-complete. As shown in Figure 2.1, the time complexity of these problems is $O(c^n)$ and the space complexity is $O(n)$. [Jansen and Sijstermans, 1989, 271] [Brassard and Bratley, 1988: 324, 336-337].

2.3 *Parallel Architectures*

The most common computer in use today is a serial machine which physically performs one task at a time. Each task must be performed in a definite order with one task following the other in sequence. In contrast, a parallel computer can perform any number of different tasks at the same time limited only by the number of processors available to perform the tasks. One useful comparison of differences in the implementations of an algorithm on serial and parallel machines is speedup. According to Miller and Penke, speedup is defined as the ratio of the time the algorithm takes to run on a serial computer versus the time the algorithm takes to run on a parallel computer. The formula for speedup is

$$S = T_{serial}/T_{parallel}$$

Another comparison between serial and parallel computers is in the area of efficiency. Efficiency is defined as speedup per processor in the parallel system [Miller and Penke, 1989: 133]. Thus,

$$E = S/P$$

where P is the number of processors in the parallel system. Ideally, the speedup is a linear function and the efficiency is constant. For reasons discussed later, this is very seldom the case.

2.3.0.1 Types of Parallel Computers One of the main differences among the categories of parallel computers in use today is how their memories are organized. As described by DeCegama, shared memory computers have one large block of memory which all the processors can access. Communication between processors is accomplished by one processor placing information in a memory location and other processors reading that location. The other system of memory storage is a distributed memory computer. In this architecture, every processor has its own memory which only it can access. Communication between processors is accomplished by passing messages

between the processors [DeCegama 1989: 18-23 and 62-64]. This research concentrates only on the distributed memory computer.

Another main difference among parallel computers is the communications network used to pass information. One method is to connect all the processors and memory to a bus. This allows a few lines to completely connect all processors and memory. The main disadvantage of a bus architecture is of the limited bandwidth of the bus. According to DeCegama,

> "... if the number of processors is large (from 50 to 100 processors with present technology), the delays due to bus contentions for interprocessor communications and global memory accesses are increasingly unacceptable, and performance degrades rapidly"

[DeCegama, 1989: 192].

The other communications network is a *switching* network of interconnecting lines. Processors and memory cells are directly connected only to a fraction of the total number of processors available. DeCegama provides an explanation for many types of switching networks including the crossbar network, the wraparound mesh network, the shuffle-exchange network, the SW-Banyon, and the generalized cube [DeCegama, 1989: 199-253]. Messages between processors not directly connected must be routed, or switched, by intermediate processors or switching units similar to those used in telephone switching circuits.

DeCegama also categorized parallel computers by the way instructions and data are processed. Only the two most common categories, single instruction multiple data (SIMD) and multiple input multiple data (MIMD) are discussed here [DeCegama, 1985: 63-65].

- A SIMD computer has multiple processors with each processor performing the same instruction on different data at the same time. All instructions are executed synchronously on all processors.

- A MIMD computer has multiple processors, each capable of asynchronously executing different instructions on different data sets. The processors can work independently or as a group.

2.3.1 Hypercube Architecture The hypercube is a distributed memory MIMD computer. Each processor has its own local memory and information is disseminated by passing messages between processors. Each processor can work independently on its own data, or work as a group on shared data.

Hayes and Mudge describe a hypercube architecture as a generalization of the 3 dimensional cube graph to an arbitrary number of dimensions. Just as a 3 dimensional physical cube has 2^3 vertices, so an n dimensional hypercube has $N = 2^n$ nodes. Each vertex has n nearest neighbors. Figure 2.2 shows four examples of the connections of a hypercube for different values of n. This topology guarantees that any 2 vertices are no more than n links apart. Therefore, the time to communicate between any 2 vertices is $log_2 N$ in the worst case [Hayes and Mudge, 1989: 1829-1830].

2.3.2 Granularity Grain size, or granularity, is used in parallel computers to describe the relative size or frequency of an event as compared to other events of the same type. DeCegama categorizes granularity into 2 broad areas: system and application granularity. System granularity is used to describe attributes of the hardware and physical configuration while application granularity describes the characteristics of the particular problem being solved [Decegama ,1989: 8-9]. Both of these granularities influence the effectiveness of a parallel program.

2.3.2.1 System Granularity System granularity is classified into three grain sizes: coarse grain, medium grain, and fine grain. Generally, a coarse grained multi-processor computer has a small number of large, complex processors while a fine grain multi-processor computer has a large number of small, relatively simple processors. As the name implies, a medium grain

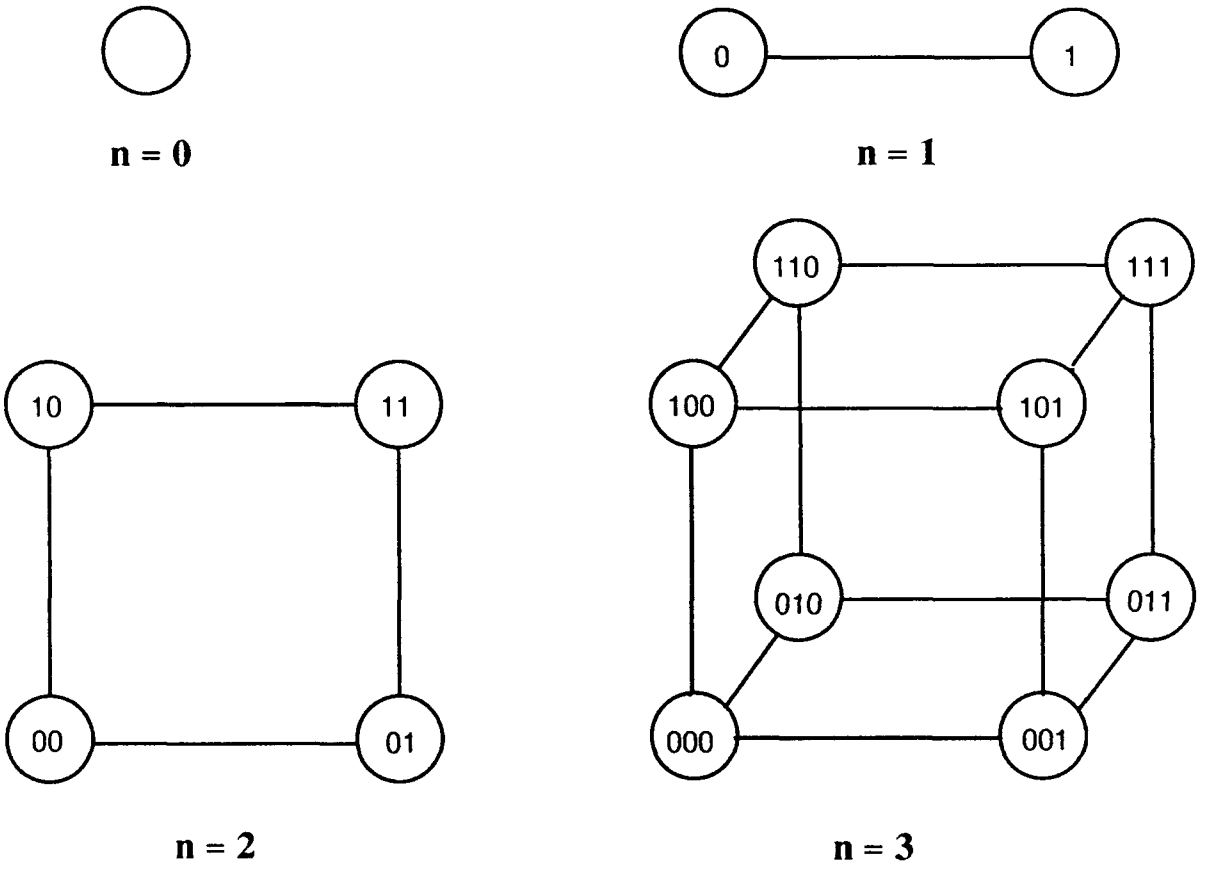


Figure 2.2. Hypercube Dimensions

computer is in between coarse and fine grain in both the number of processors and the complexity of the processors used. Most of the commercial parallel computers in use today are medium grain [Decegama ,1989: 8-9]. The definition of what is fine or course grain is subjective and constantly changes as the technology changes.

2.3.2.2 Application Granularity DeCegama divides application granularity into event and task granularity. A task is defined as a program segment which must be executed sequentially. Task granularity is the average amount of computation performed by each task of the program. Event granularity is a measure of the average amount of computation performed by the processors between events of a certain type. For example, communication granularity is the amount of computation between message events. Other common event granularities are synchronization, heuristic, and voting [Decegama ,1989: 8-9].

Like system granularity, event granularity is also measured by relative comparisons between the number of events. Coarse grain events have relatively large amounts of computations between events and fine grain events occur relative frequently.

2.3.2.3 Granularity Tradeoffs There are overhead costs associated with event granularities. Processing the event, calls to operating system or other functions, communication between processors, and resource contention are all overhead which reduce the amount of time spent solving the problem. Increasing the event granularity decreases these costs. However, many algorithms require fine grain events to operate efficiently. For example, information calculated by one processor might be required by all processors. Delaying the communication of the data could result in longer execution times for the algorithm.

Increasing task granularity decreases the number of tasks in a computation. This decreases the overhead associated with task creation and termination, but might introduce other costs. A small number of large tasks might make it harder to balance the work load between the processors

because the tasks can not be divided into smaller work. This could result in idle processors and a longer execution time for an algorithm.

Whenever designing a parallel algorithm, both system and application granularity must be considered. Tradeoffs between event grain and the associated overhead costs must be carefully weighed to obtain the optimal performance of the algorithm.

2.4 Parallel Search Issues

Architectures discussed previously are applicable to solving both serial and parallel search algorithms. However, there are issues that pertain only to parallel computers. Communication of global variables to all processors and distributing the work load evenly among the processors are the main concerns addressed.

2.4.1 Global Variable Communication Since this research covers distributed memory computers only, each processor has access only to the information stored at the local location. If a global variable changes, this new value must be communicated to all the processors. Processor time spent communicating subtracts from the time spent solving the problem. Jansen and Sijstermans recommend partitioning the processors into groups that have only one master processor communicating outside the group. This reduces the number of unnecessary communications between processors because the master validates the new global data before transmitting it inside the group or to other groups. [Jansen and Sijstermans, 1989: 275]. Another method is to transmit new global variables to all processors at the same time. Still another method is to wait until certain control points in the algorithm before transmitting global information. This last method reduces communication, but at the cost of possibly expanding unnecessary states in the search graph associated with the NP-complete problem.

2.4.2 Task Allocation Methods There are two ways to allocate new tasks to the processors: static and dynamic. Static allocation is done *a priori*. Dynamic allocation assigns processors to a problem as new children or sub-problems are generated. NP-complete problem solutions are probabilistic in that the order in which branches of the search graph are explored can not be determined. Also, the amount of work in each branch or sub-branch of the search graph can not be determined. Therefore, sub-branches generated by states, or sub-problems, are generated in an unpredictable fashion. Static allocation of only certain branches of the search graph to particular groups of processors would lead to processors which finish early being idle until the last processor finishes. Dynamic allocation allows new sub-problems to be assigned to processors with few or no problems waiting to be run. This allows all the processors to be active approximately the same length of time.

2.4.2.1 Centralized Versus Distributed List Dynamic allocation has two methods for allocating tasks. The first, centralized list (CL), keeps the listing of all the sub-problems generated in one processor called the master. The processing of the sub-problem is done in all the other processors called slaves. The advantage of the CL is the global "best" sub-problem is always assigned to the next available processor. A disadvantage of the CL is when a slave finishes or generates a sub-problem, it must contact the master to insert the sub-problem or to receive its next problem to work. This requires two communications for each initiation of a sub-problem. According to Quinn, adding additional processors to the system causes a linear increase in the communication overhead of the parallel algorithm. Also, the master is involved in all of these communications and it can become a bottleneck for the system. When messages begin to back up at the master, slaves become idle waiting for a response. Eventually, the communication overhead to the master becomes the dominant computational factor and adding more processors to the problem can actually increase the execution time of the algorithm [Quinn, 1990: 385]. The advantage of always assigning the best sub-problem to a processor is greatly outweighed by the cost of communication and waiting!

The other approach to dynamic allocation of sub-problems is the distributed list (DL). In this approach, each processor maintains a list of sub-problems waiting to be worked. Therefore, when the processor completes or generates a task, all communication is within the processor and no message is passed to another processor. This eliminates the bottleneck of having to communicate twice with the master processor when a task is completed.

One problem with the DL method is not search all problems generate the same amount of sub-problems. Therefore, to keep processors from being idle while sub-problems are still waiting to be run on other processors, a method of load balancing must be implemented. Several different load balancing algorithms and their performance are discussed by Quinn [Quinn, 1990: 385]. The choice of which sub-problem to keep and which one to send to another processor and when to balance the loads greatly affected the efficiency of the search algorithm. While load balancing adds to the communication overhead, the benefit of reduced processor idle time greatly outweighs the extra cost in lost computation time due to communication [Ma and others, 1988: 1507-1510].

2.4.3 Limits on Speedup and Efficiency Many other factors can limit the speedup and efficiency of a parallel algorithm. Since almost all programs have statements or routines which don't depend on values from other parts of the program, recognizing and efficiently exploiting this parallelism is essential to produce the maximum speedup possible [Hayes and Mudge, 1989:1834]. Another area that reduces efficiency is having the processors idle until enough tasks are generated. Changing the method of generating the tasks depending on where the algorithm is in the search can minimize this problem. A final problem is that parts of the algorithm cannot be parallelized. Starting, terminating, and certain other procedures in an algorithm are inherently sequential and cannot be done in parallel. All these items decrease the speedup of an algorithm.

2.5 *General Search Techniques*

A search problem can be represented by a search graph with the root of the graph representing the complete problem to be solved. Each child node represents a subproblem of the parent node and represents inclusion of one or more constraints to the problem [Quinn, 1990: 384] [Hayes and Mudge, 1989: 1838]. To find an optimal solution, every node, or state, of the graph must be explicitly or implicitly checked to see if it is a solution. As described in Chapter I, the state space can be extremely large making it impossible to explicitly check every node. The main difference between the search techniques described below is the order in which the nodes are selected to be investigated or explored. The rest of this section describes various search techniques: the greedy method, uninformed search, backtracking, branch and bound, and best first search. While this is not a definitive list of search techniques, most other techniques which provide an optimal solution are some combination of the techniques described.

2.5.1 Greedy Method In some greedy algorithms, enough information is known about the problem to always ensure the search is on the path to the optimal solution. Since at each stage of the search the "best" node is selected, only the minimum number of nodes are expanded. Other greedy algorithms, such as hill climbing, expand the best node at each level but retain no state information on parent or sibling nodes. This can result in an algorithm returning a local, not absolute, minimum as a solution. [Pearl, 1988: 35]. Examples of greedy algorithms include the minimum spanning tree algorithm and Dijkstra's algorithm to solve the shortest path problem [Brassard and Brantley, 1988: 80-87]. These problems are not NP-complete since the algorithms can solve them in polynomial time.

2.5.2 Uninformed Search Another name for uninformed search is the "brute force" method. This technique expands every node without considering if it is on a solution path or not. If a solution is found, the algorithm stops. There are two main variations in uninformed search; depth first search (DFS), and breadth first search (BFS).

2.5.2.1 Depth First Search (DFS) Depth first search works by always generating a child node from the most recently expanded node. This continues until a solution is found or the next state to be generated is not feasible. Thus, priority is given to expanding nodes at deeper levels of the search graph. See Figure 2.3 for the following example. The search starts at the root node, R, and continues down the path from node 1 to node 2 ending at node 4. At each level only one node is expanded before going on to the next level. If the solution is not on the path expanded, no solution is found [Pearl, 1985: 36].

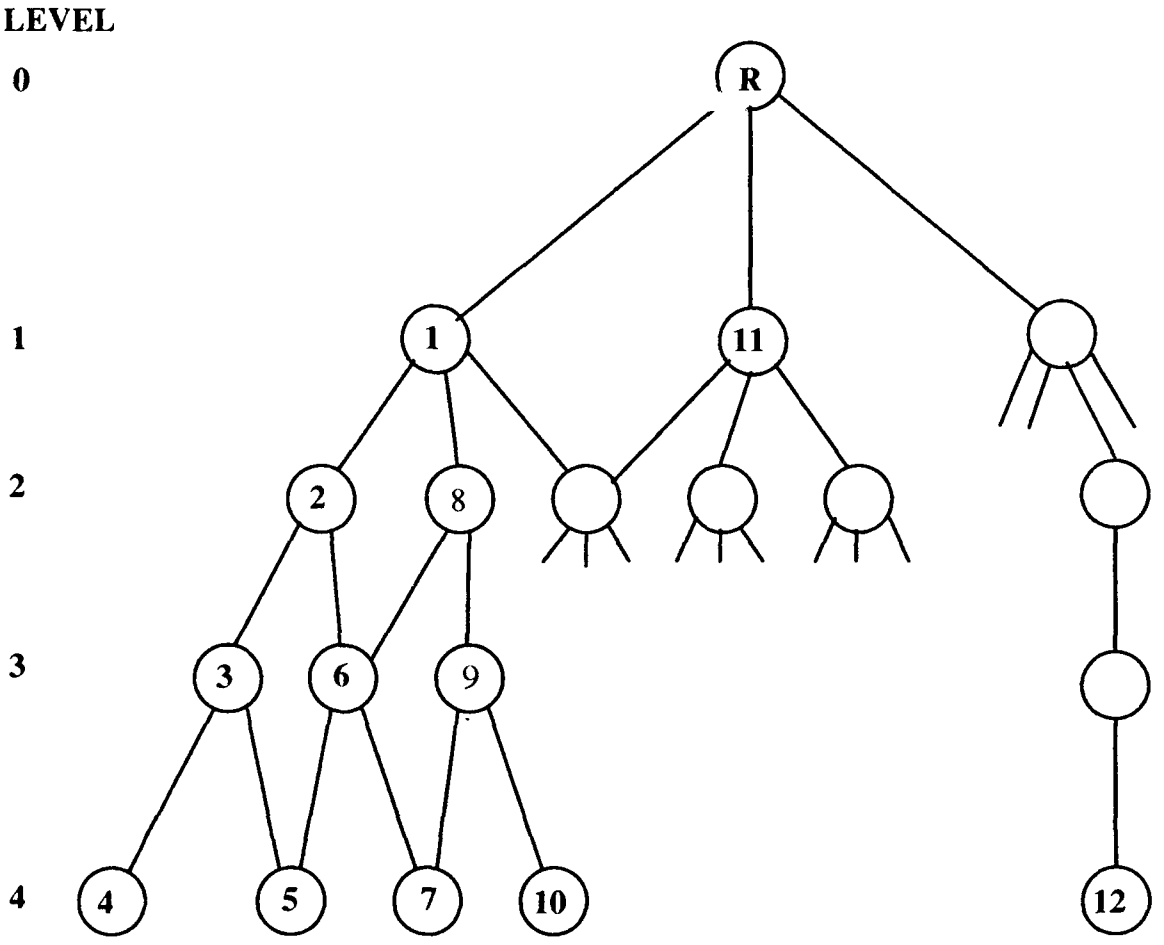


Figure 2.3. Depth First Search

A variation of DFS proposed by Korf is depth first iterative deepening (DFID). This algorithm begins at the root node, level 0, and performs a DFS to level one, expanding all nodes to this level.

If a goal node was not found, discard all nodes generated and start over at level 0 and perform a DFS to level two. Continue discarding nodes and performing depth first searches until a solution is found. One disadvantage of this algorithm is that it performs wasted computations by discarding and generating the same nodes repeatedly. Korf claims that for large problems, the number of nodes expanded asymptotically approaches the number of nodes for regular DFS. He states that since almost all work is performed at the deepest level of the graph, most nodes are only expanded a few times. However, Korf assumes the cost of generating a node is cheap while the cost of storage is high. This is not always the case. One advantage of DFID is since every node at each level is expanded, it finds the shortest path solution. The other advantage is that only small amounts of memory are required since only the path to each node and the cost to reach the node is stored [Korf, 1985: 98-106].

2.5.2.2 Breadth First Search (BFS) In contrast to DFS, BFS assigns a higher priority to expanding nodes at a higher level in the search graph. The list of nodes to be expanded can be stored in a first-in-first-out (FIFO) queue. See Figure 2.4 for the following example. The search starts at node R at level 0. At level 1, nodes 1, 2, and 3 are expanded before going on to level 2. Since it expands every node at each level before continuing down to another level of the graph, the first solution path found by BFS is the one with the shortest path [Pearl, 1985: 42] .

2.5.2.3 Backtracking In the DFS once the deepest node on the path was reached, the search ended even if no solution was found. To continue searching for another, possibly better, solution the algorithm must "backtrack" back up the path to a higher level. At each higher level, the node is checked for any unexpanded child nodes. If a child node is found, a new DFS is started down that path. If no unexpanded child node was found, the algorithm backtracks to the next higher level. This continues until all paths have been searched. In this case, the list of nodes to be expanded can be stored in a last-in-first-out (LIFO) queue. In Figure 2.3, if node 4 was not the solution , the algorithm backtracks to node 3 and checks to see if it has any unexpanded child

nodes. In this example, node 5 would be expanded next followed by the nodes in increasing order. Nodes at a deeper level of the search graph are still given a higher priority for expansion than nodes at a higher level [Pearl 1985: 36-41].

2.5.2.4 Branch and Bound To keep from having to explicitly explore every path of the search graph, additional information must be stored. For example, if a maximum cost is known after the first branch of the tree is explored, this cost can be stored and used to limit the search down any other branch. If a lower maximum cost is found in another branch, this new value is stored as the new maximum cost. Also, if additional information is known about the particular problem being solved, a heuristic function can be used to calculate the cost of continuing to search down a particular branch of the tree. An example of this function is the cost of reaching a node plus a conservative estimate of the cost to the solution. If this value is greater than the known maximum cost, or lower bound, then the search would not continue down this branch. Instead the algorithm would jump, or branch, to the next node at the head of the stack or queue waiting to be explored. In this way, all of the nodes of the tree do not have to be explicitly explored.

Bosworth and others characterize a branch and bound algorithm into the following four main parts:

1. Expansion procedure — A method to create a node's children
2. Selection procedure — A heuristic such as DFS or BFS to decide the order in which the nodes are expanded
3. Bounding rule — Does the cost to reach a particular node plus the estimated cost to completion for that node equal or exceed the global best cost.
4. Termination rule — Determination of whether the node represents a solution. When searching for an optimal solution, termination is delayed until all nodes have been evaluated either explicitly or implicitly.

[Pennington and others, 1988: 241]

Using a combination of backtracking, branch and bound, and DFS an optimal solution to the search graph can be found quicker than using just DFS. An advantage to this method is it requires less memory storage than other search techniques since only the paths with solutions are stored. A disadvantage of this method is the algorithm can take longer than BFS. For example, in Figure 2.3 the search works from top to bottom then left to right. If the solution was in the path containing node 12, most of the nodes expanded were not on the solution path . However, if an optimal solution is required, all branches of the graph would have to be searched, or bounded, to validate that the best solution was found [Korf, 1985: 99].

In contrast, using branch and bound with BFS provides a solution quicker since it finds the solution with the shortest path. The time complexity is also at least $O(c^{f(n)})$. However, BFS requires more memory storage because all nodes at each level are expanded before going to the next level. Since this requires all paths of the search graph to be stored *until a solution is found*, the memory requirements could be the entire search space. For NP-complete problems, this is at least $O(c^{f(n)})$ where c is a constant and $f(n)$ is a function of the number of inputs into the problem. When searching for an optimal solution, BFS continues searching the graph until all branches have been explored. Only when the cost of continuing down a path exceeds the current maximum value is a path removed from memory. Korf points out that many times the memory required to be stored exceeds the memory of the computer. When this occurs, the problem is not solvable with these techniques [Korf, 1985: 100-102].

2.5.3 Best First Search Like the DFS with branch and bound, best first search uses an heuristic to calculate the estimated cost to find a solution down a certain path. Unlike DFS which expands the best node from the most recently expanded node, best first search expands the best node in the entire graph. Using the heuristic information, best first search focuses the search down the path which provides the best chance of producing a solution.

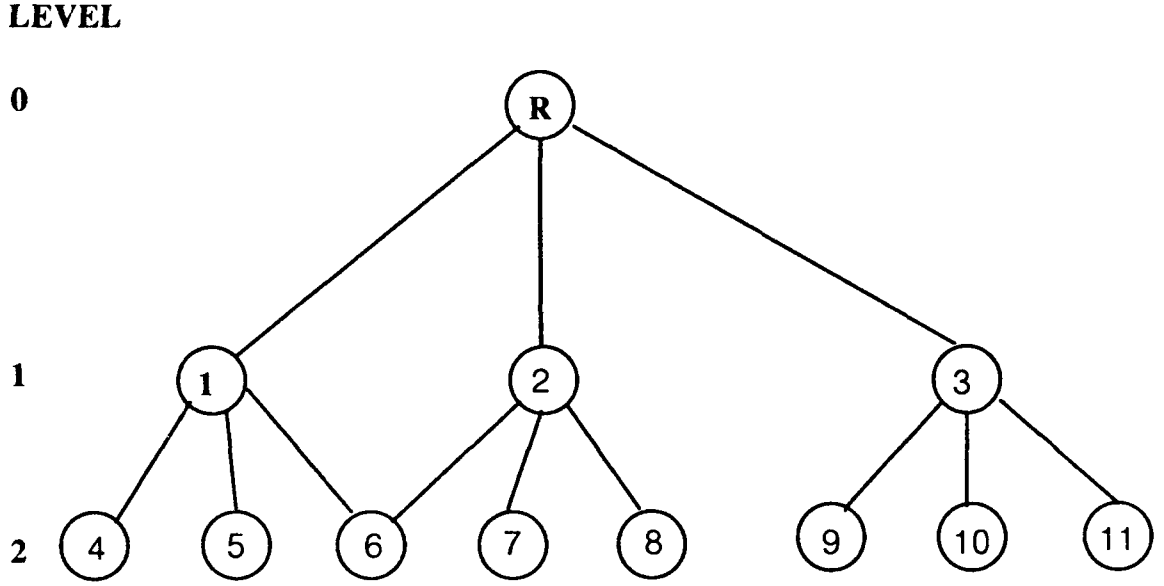


Figure 2.4. Breadth First Search

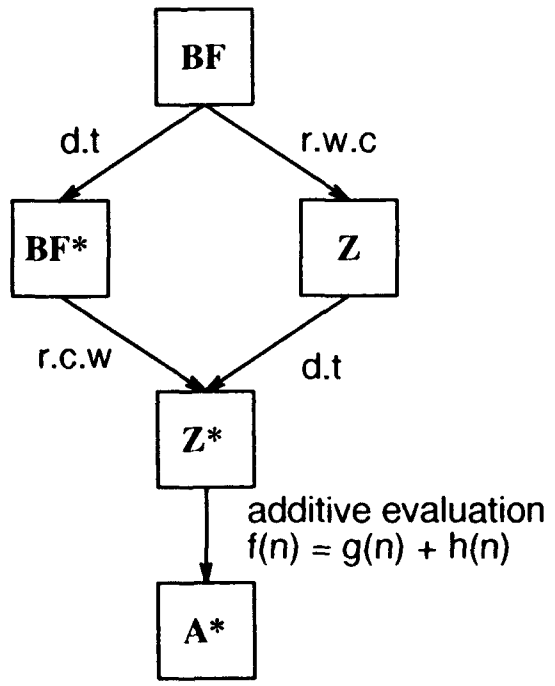


Figure 2.5. Hierarchical Diagram of Best First Algorithms

2.5.3.1 Hierarchy of Algorithms Pearl describes a hierarchy of best first algorithms based on when the algorithm is terminated and how the cost of a node is calculated. This hierarchy is shown in Figure 2.5. In Figure 2.5, d.t. stands for delayed termination, * signifies an optimal solution is found if one exists, and r.w.c. stands for recursive weight function. Pearl defines a recursive weight, or cost, function as follows:

> A weight function $W_G(n)$ is *recursive* if for every node n in the graph
>
> $$W_G(n) = F[E(n) : W_G(n_1)mW_G(n_2), \ldots, W_G(n_b)]$$
>
> where $n_1, n_2, \ldots, n_b$ are the immediate successors of n. E(n) stands for a set of local properties characterizing the node n. F is an arbitrary combination function, monotonic in its $W_G()$ arguments.

Basically, if the weight of a node is recursive, the weight is a function of the weights of the nodes in its path.

An optimal solution can be found using best first search by combining it into an algorithm with branch and bound and delaying termination of the algorithm until all branches of the graph have been evaluated.

*2.5.3.2 A** One algorithm used to calculate the estimated cost to a solution is an additive function of the form

$$f(n) = g(n) + h(n)$$

where g(n) is the cost of the path from the root node to node n and h(n) is the estimated cost from node n to the solution [Pearl, 1985: 75]. As Figure 2.5 shows if this heuristic function is used with best first algorithm and termination is delayed to search for an optimal solution, the algorithm is called additive optimal, or A*.

Pearl defines a heuristic as *admissible* if

$$h(n) \leq h*(n)$$

where h(n) is the estimated cost to completion and h*(n) is the actual cost to completion [Pearl, 1985: 77]. If an admissible heuristic is used in the A* algorithm, you are guaranteed to always find an optimal solution if one exists [Korf, 1985: 103].

2.5.3.3 A Variations* Korf suggests using a variation of the depth first iterative deepening algorithm with the A* algorithm called IDA*. At each iteration, perform a DFS, bounding the path when the f(n) value exceeds a given threshold. The initial threshold is the estimated completion cost of the root node. The threshold used for the next iteration is the minimum cost of all values that exceed the current threshold. The algorithm ends when all nodes have been explored or unexplored nodes exceed the cost of the threshold.[Korf, 1985: 103]. Another variation of IDA* is to set the threshold by the number of levels expanded. For example, on the first iteration expand all nodes to level 3. Then on the second iteration, begin at level 0 and expand all nodes to level 6, and then the third iteration would begin at level 0 and expand all nodes to level 9.

For example, Figure 2.6 shows the partial search space for a problem using cost as the threshold to determine which nodes are expanded. The root node generates all of its children with estimated cost less than or equal to the estimated cost of root node. On the first iteration, all nodes at level 1 are generated and the minimum cost is now 110. On the second iteration, level 1 is generated again, but only node 1 is expanded. Node 2 is also expanded since its estimated cost is still equal to 110. The new threshold value is now 111 from node 6. Each successive iteration generates level 1, but only expands nodes with cost not exceeding the threshold value. The third iteration expands nodes R,1,2,6; generates nodes 3,5,7,8,10,11; and the new threshold is 112. The fourth iteration expands nodes R,1,2,3,5,6,7,8, and generates nodes 4,9,11,12, and 13. This continues until a solution node is found which becomes the new threshold and the process continues until all branches have been investigated.

Korf claims most of the work by IDA* generating nodes is performed at the bottom of the tree so the number of nodes generated asymptotically approaches the number of nodes generated

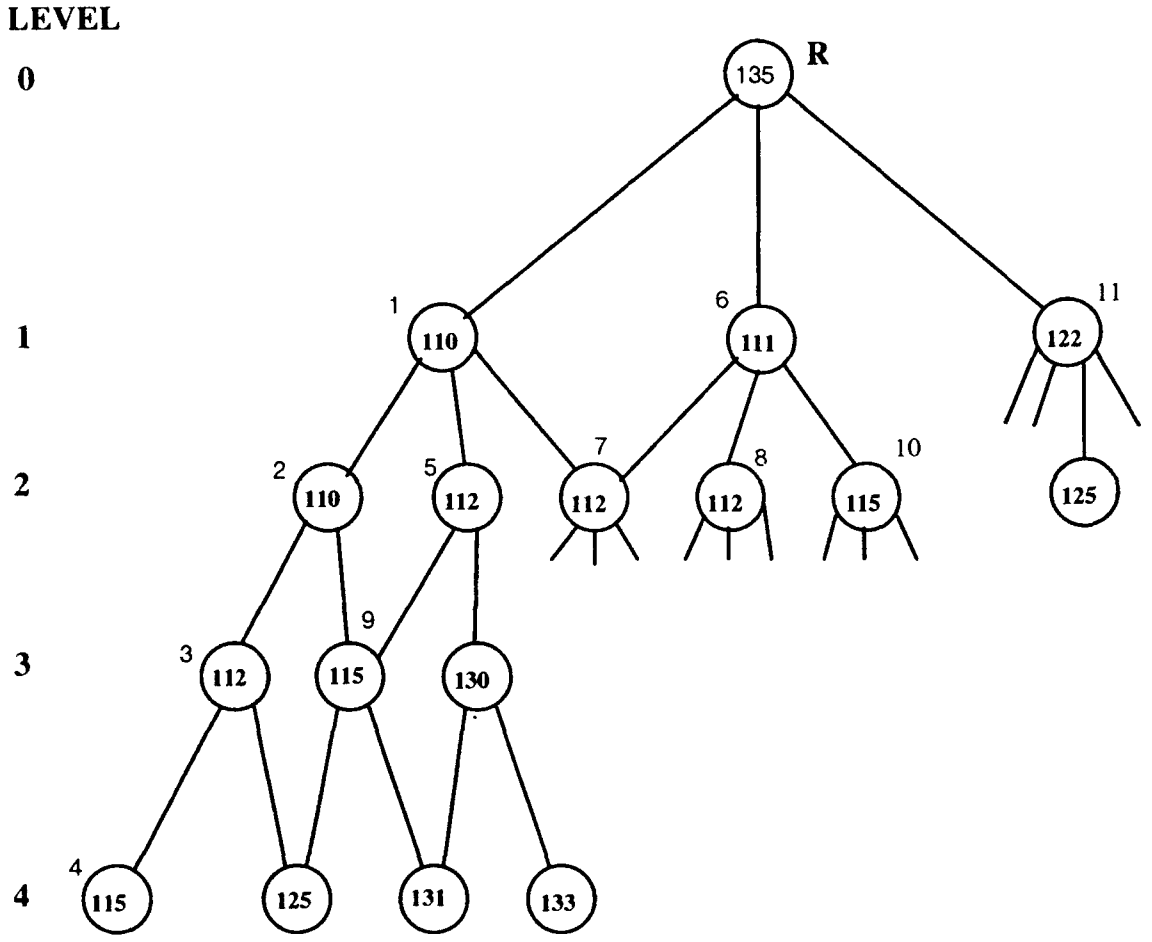


Figure 2.6. Iterative Deepening A*

by A*. Since only the path from the root node to the solution node must be stored, he claims to get A* speed with depth first search memory requirements [Korf, 1985; 106].

Cvetanovic and Nofsinger suggest another A* algorithm using what they term *Continuous Diffusion*. This algorithm is used on distributed memory parallel computers using a distributed list algorithm. This algorithm performs a parallel A* search, but after expanding a set number of nodes, processors then exchange a certain number of nodes from their list to be expanded with their nearest neighbors. This keeps processors from expanding nodes with higher costs while a neighbor has nodes with much lower cost. The nodes with the lowest cost *diffuse* from processor to processor insuring the best nodes are being expanded. The idea is to keep the local distributed list implementation as close to a centralized list implementation as possible. Using this method, they claim to expand a much smaller number of nodes than IDA*. See the section on Parallel Architecture for an explanation of distributed memory and nearest neighbor [Cvetanovic and Nofsinger, 1990: 87].

2.6 Summary of Search Algorithms

This section is a summary of the search algorithms presented. Table 2.6 is only a generalized description of the algorithms. Specific problems and implementations can greatly affect the memory and time utilization of the algorithms. For example, all parallel versions of A* are greatly affected by whether the list of work to be done is maintained on a centralized or distributed list.

Table 2.6 is a summary of some important papers about search algorithms and NP-complete problems.

SUMMARY

Algorithms to solve NP-complete on serial computers are well known. However, NP-complete algorithms implemented on parallel computers have been studied only in the last decade, and many fundamental questions remain unanswered. New techniques for load balancing and communicating

Parallel Search Algorithms			
ALGORITHM	MEMORY	TIME	COMMENTS
Depth First	Requires little memory, Best feature	Varies greatly depending on where in search graph a solution is located. Can require prohibitive amount of time.	Branch and bound and backtracking can greatly reduce time
Breadth First	Can require prohibitive amounts of memory	Varies greatly depending on where in search graph a solution is located.	Branch and bound and backtracking can greatly reduce time and memory required.
Best First	Can require prohibitive amounts of memory. Uses less than Breadth First	Solutions are consistently quickest, but can be longer than Depth First	Characteristics depend on the variation used
A*	Same as Best First	Same as Best First	
IDA*	Memory the same as Depth First	Claimed to be same as A*	Still undecided issues on relative speed, especially in parallel version
Continuous Diffusion A*	Same as Best First	Claimed to be better than IDA*, definitely better than A*	Parallel version only Dispute about relative merit compared to IDA*

Table 2.1. Memory and Time Comparisons of Search Algorithms

Parallel Search Algorithms		
Year	Investigators	Description
1976	Korf	Theoretical investigation of Depth First Iterative Deepening
1988	Pennington, Bosworth, Wheeler, Stiles and Raghuram	Branch and Bound Algorithms for Distributed Database Networks
1989	Jansen and Sijstermans	Parallel Branch and Bound Algorithms
1989	Hays and Mudge	Overview of hypercube architectures with 2 examples of use
1989	Miller and Penke	Parallel Traveling Salesman Program and factors which affect speedup
1990	Li and Wa	Good discussion of anomalies in parallel search algorithms
1990	Quinn	Theoretical and measured evaluation of different load balancing algorithms for the hypercube
1990	Cvetanovic and Nofsinger	Different hypercube load balancing using *Continuous Diffusion*

Table 2.2. Applications and Implementations of Search Algorithms

global variables are among the main areas of research. Study is also underway on how parallel algorithms work and ways to increase the speedup.

III. Methodology and Design

3.1 Introduction

This chapter discusses the methodology used in this research and the preliminary design of a parallel A* algorithm. The methodology is described in section 3.2 and the preliminary design in section 3.3. Complexity analysis of the design is provided in section 3.4.

3.2 Methodology

Designing and implementing a complicated sequential algorithm can be extremely difficult. The additional complexities of parallel algorithms discussed in Chapter II accent the requirement for a systematic approach to designing parallel algorithms. For all algorithms used in this research, the first step was to develop a thorough understanding of the problem. Chapter II provides the background for understanding NP-complete problems, search algorithms, and computer architecture.

Next, preliminary and detailed algorithms were designed using a top-down approach. Each function and data structure was designed to allow modification or incorporation into other algorithms. Many of the functions and algorithms are designed to be run sequentially on parallel processors. This allows use of a personal computer using BORLAND C++ for the initial implementation and debugging.

Then the sequential algorithms and functions were combined and implemented on the Intel iPSC/2 hypercube and tested to validate proper execution. Many test cases and examples were used to attempt to test all function. Some functions were of a size or importance to be tested completely. For example, the operation on the queue which stored the work to be performed are critical to the proper operation of the algorithm and could be exhaustively tested.

Finally, data was collected and analyzed to evaluate the performance of the algorithm. Performance metrics discussed in the next section were the main measures of performance. The data

was evaluated trying to understand why changes to the algorithms produced these results, how do the different algorithms compare, and which algorithm would be better for different problems.

3.3 Metrics

3.3.1 Speedup Many metrics can be used to measure the effectiveness of a parallel algorithm. Chapter II gave the definition of speedup and efficiency of parallel algorithms as:

$$S = T_{serial}/T_{parallel}$$

and

$$E = S/P$$

These are good metrics when trying to compare the total time to run different algorithms on the same type of computer. However, it can be difficult comparing run times from different types of computers because of different clock rates, communication schemes, memory schemes, and many other factors. Therefore, small variations in run times on different computers is not important.

Normally, the best speedup achieved is linear. For example, if the time to run the sequential program is 100 seconds and the parallel program time is 50 seconds using 2 processors, the speedup is 2. Ideally, if 4 processors are used, the parallel time would decrease to 25 seconds for a speedup of 4. Thus, in this example the speedup is linearly proportional to the number of processors used.

However, Miller and Penke describe many limitations on the achievable speedup. First, the startup and termination of all algorithms are by nature sequential and cannot be parallelized. Another limitation on the speedup is the extra work performed by the parallel search algorithm. As described in the next section, parallel search algorithms perform extra work as compared to the sequential algorithm. Also, there are time costs associated with communication and/or memory

contentions in parallel computers not found on sequential computers. All of these problems decrease the amount of speedup achieved [Miller and Penke 1989: 133].

Sometimes the speedup is greater than linear. This is normally considered an anomaly and not true speedup. Li and Wa provide the following reasons or conditions which can result is super-linear speedup:

1. There are multiple solution nodes. This can allow the parallel search algorithm to find a solution before the sequential algorithm.
2. The heuristic function is ambiguous and allows for selection of more than one path.
3. The rule used to eliminate nodes isn't consistent with the heuristic function.
4. The tree structure of the search space causes nodes not expanded in the sequential algorithm to be expanded when using multiple processors.
5. The feasible solutions are not generated in the same order when different number of processors are used.

As they point out, different combinations of these conditions cause the tree to be searched in different orders depending on the number of processors used [Li and Wah, 1990: 21-29]. If the parallel algorithm has super-linear speedup only on particular data sets, then these cases are probably anomalies. However, if the parallel algorithm consistently has super-linear speedup over all data sets , the sequential algorithm is not designed well and can be improved.

3.3.2 Nodes Expanded Another metric which is less dependent on the type of computer used is the number of nodes expanded by the algorithm. Using Figure 2.3 as an example, if node 4 was a solution (goal) node, then the minimum number of nodes would be expanded. However, if node 10 was the goal node, then expansion of all nodes not in its path was wasted work. Obviously, the more efficient algorithms expand fewer nodes not on the solution path.

Figure 3.1 represents the total search space for a given problem with the numbers representing locations of optimal solutions.

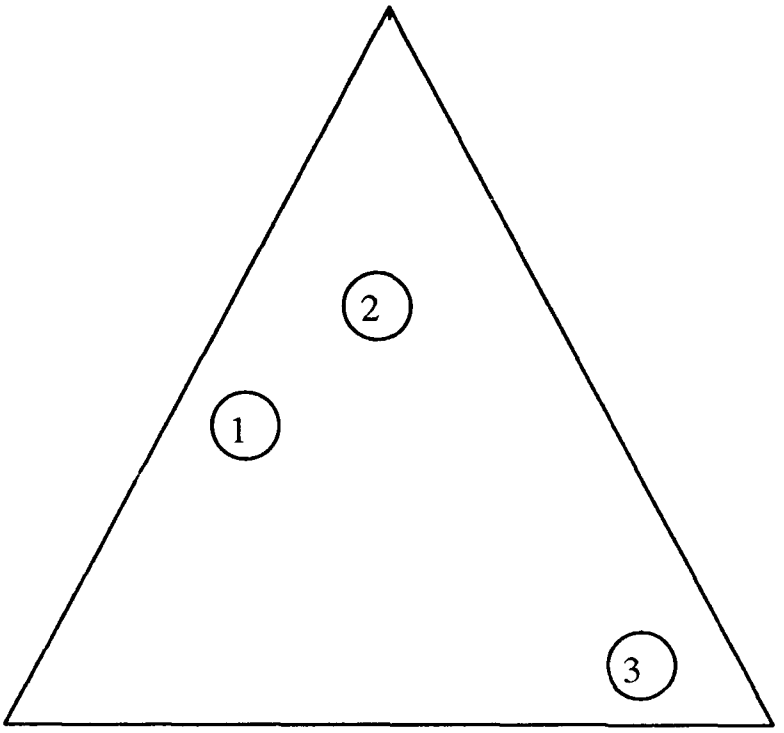
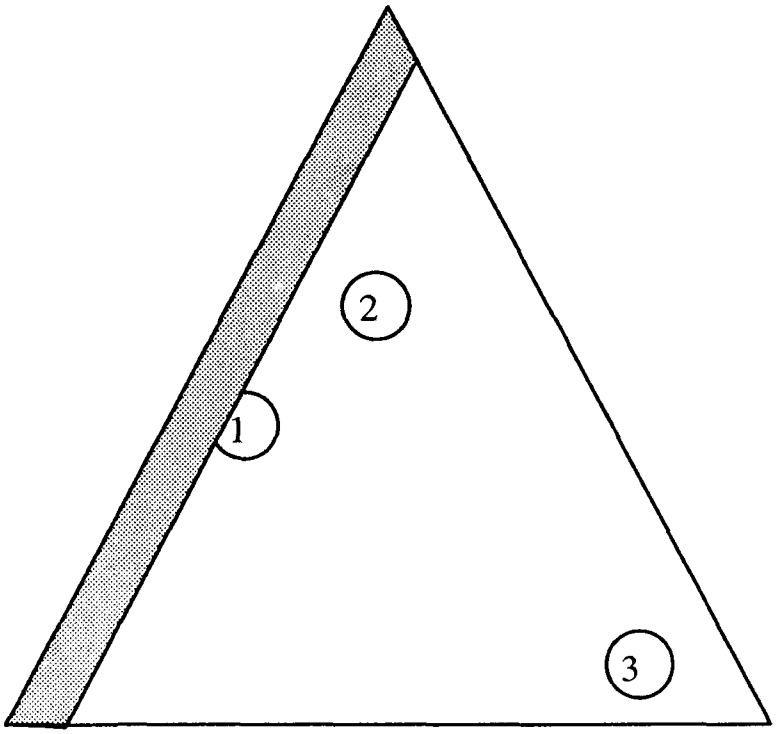


Figure 3.1. Total Search Space

Figure 3.2 shows the portion of the space searched by DFS to find solution 1. If 3 was the only solution, practically the entire space would be searched even using a branch and bound DFS algorithm. If a "near" optimal solution was found early in the search in Figure 3.2, then most of the search space could be implicitly checked without having to expand the nodes. This could save time, but there is no way to guarantee a near optimal solution will be found early in the search.



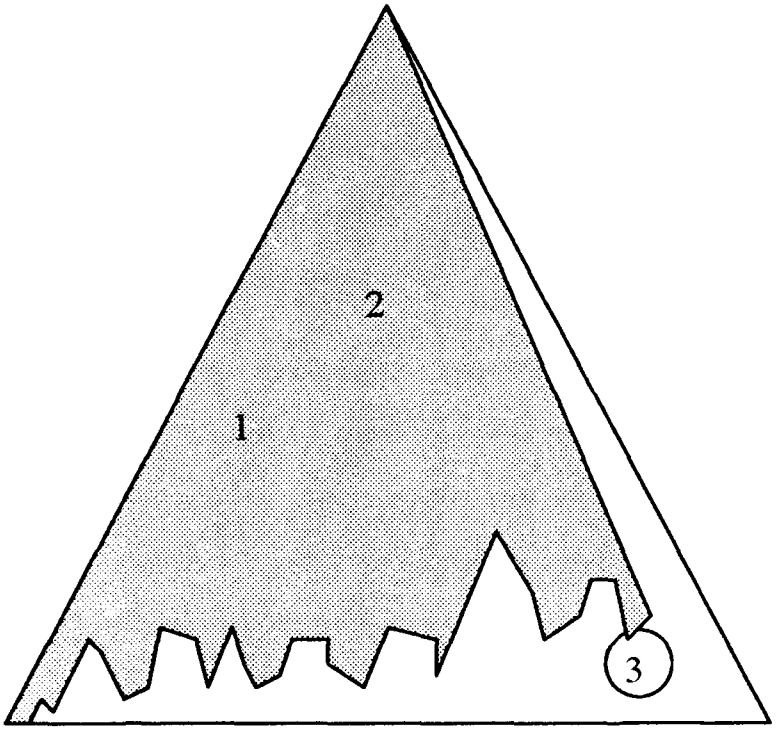
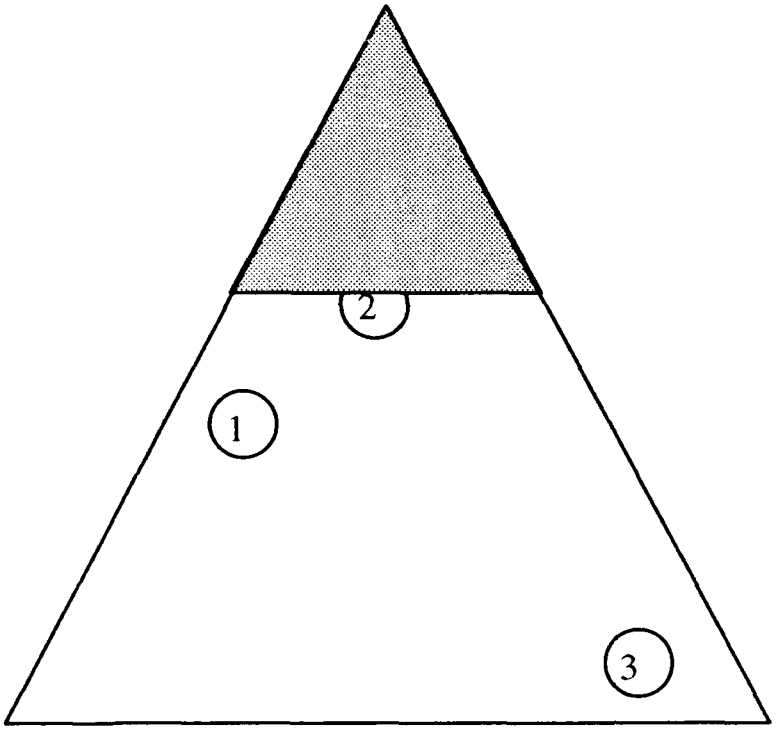


Figure 3.2. Search Space For Depth First Search

Figure 3.3 shows the search space searched to find a solution using BFS. Notice solution 2 is found first with very little of the search space explored. However, nearly the entire search space must be explored if 3 is the only solution. Even with backtracking and branch and bound algorithms, the program might take too long to run in both of these cases.

A better way to explore the search space is shown in Figure 3.4. No matter where a solution is located in the space, the search concentrates on the path to it. While this seems obvious, the hard part is designing an algorithm which explores the search space in this manner. As seen from these examples, the number of nodes expanded can be a "good" metric to determine the efficiency of the search algorithm.

However, a problem can arise when using just the number of nodes as the only metric. Miller and Penke state a sequential version of a search algorithm normally expands fewer nodes than a parallel version. This is because many nodes in the sequential version are not evaluated because their costs exceed the global best cost. On a parallel search, the higher levels of the search space can have many nodes which are less than the global best cost. Some of these nodes may be expanded needlessly before the global best cost is reduced. Also, the sequential version always has the complete list of *open* nodes, or nodes waiting to be expanded. Therefore, the sequential algorithm can always choose the best node to expand next. On one version of the parallel A* algorithm, the complete open list is maintained on a central processor, i.e., a centralized list, or partial lists are kept on each individual processor. This means that some processors are expanding nodes which are not the global best [Miller and Penke, 1989: 133]. Therefore, while the number of nodes expanded is less for the sequential version, the total time can be much greater. This is one reason why more than one metric should be used to evaluate an algorithm.



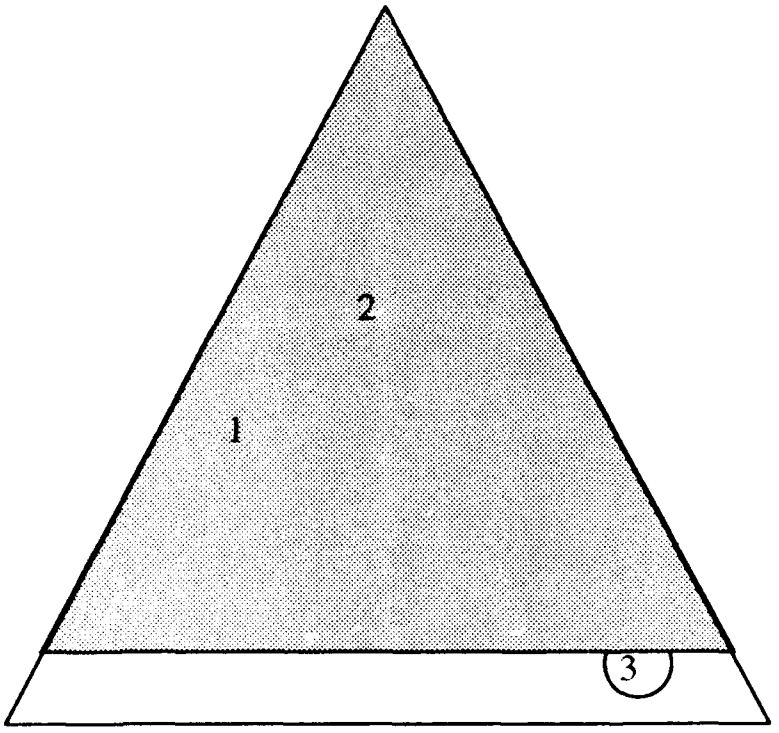


Figure 3.3. Search Space For Breadth First Search

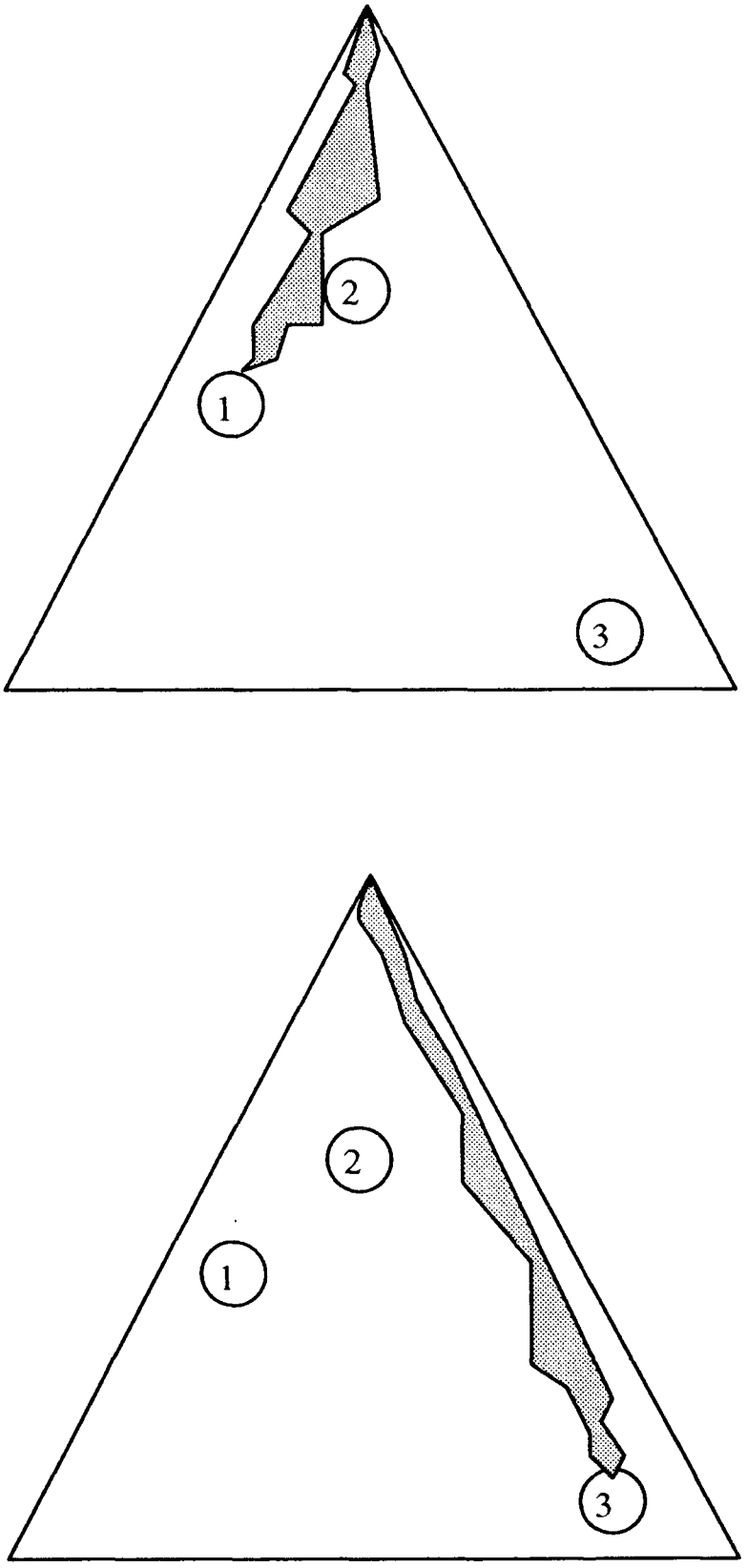


Figure 3.4. Search Space For Best First Search

3.4 *Understanding the Problem*

3.4.1 Traveling Salesman Problem The A* algorithm is a method used to determine the solution to problems requiring a search of all possible solutions. To study its characteristics and performance on a distributed memory parallel computer, a family of problems is required to be solved using the A* algorithm. For this research, I chose the traveling salesman problem (TSP). One reason the TSP was selected to use with the A* algorithm was because it is one of the most widely studied families of NP-complete problems. Since it so widely studied, the sequential implementation of TSP is well understood and there are many good sequential algorithms already developed to compare the parallel algorithm against. Also, since any NP-complete problem can be mapped to any other NP-complete problem in polynomial time, all NP-complete problems could be solved using the TSP.

The TSP consists of a graph of cities and the associated costs to travel between cities. In the TSP graph, the cities are represented by the vertices of the graph and the distances between cities by the edges of the graph. If every city has a direct path to every other city, the graph is *completely* connected. If the graph is traversed and every vertex is visited exactly once and the beginning and final vertices are the same, then the traversal is called a *tour* [Christofides, 1975: 6-9]. The goal of the TSP is to begin at an arbitrary city and complete a tour of the cities traveling the shortest possible distance [Brassard and Brantley, 1988: 103]. The cost of the tour is the sum of the costs of the edges of the tour.

If the cost of traveling between cities i and j is stored in a *cost matrix* at location ij, then the TSP can be stated mathematically as:

Minimize

$$\sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \tag{3.1}$$

subject to

$$\sum_{j \in V} x_{ij} = 1, \quad i \in V, \quad \sum_{i \in V} x_{ij} = 1, \quad j \in V, \tag{3.2}$$

$$\sum_{i \in S} \sum_{j \in V-S} x_{ij} \geq 1, \quad for\ all\ S \subset V, S \neq \emptyset, \tag{3.3}$$

$$x_{ij} = 0\ or\ 1, \quad i, j \in V, \tag{3.4}$$

where $x_{ij} = 1$ if edge $< i, j, >$ is in the solution and 0 otherwise. Equation 3.2 and 3.3 ensure that the solution is a tour and equation 3.4 eliminates the possibilities of subtours [Rottman, 1990: 97-98].

To solve the TSP, all possible combinations or paths between cities must be checked to find the optimal solution. This is accomplished by starting at an arbitrary city and adding cities one at a time to the list of cities visited. After each city is added, the list is checked to see if the cities are a tour and an admissible heuristic function is used to determine the cost of continuing down this path to completion. If the cities are a tour, their cost is compared to the best cost found so far and the smaller value is retained as the new best cost. If the cities are not a tour, then the estimated cost returned by the heuristic function is compared to the best cost. If the estimated cost is greater than the best cost, then this path is removed from the list of solution paths to be explored since its cost is higher than a solution already found.

*3.4.2 A** The A* algorithm is a specialized form of the best first algorithm. As discussed in Chapter II, using an admissible heuristic guarantees finding an optimal solution if one exists. Also discussed in Chapter II was that

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost of the path from the root node to node n and $h(n)$ is the estimated cost from node n to the solution [Pearl,1985: 75]. The sequential algorithm is described by Pearl as follows:

1. Put the start vertex s on OPEN list
2. If OPEN is empty, exit with no solution found
3. Remove from OPEN and place on CLOSED list a node n for which f is a minimum
4. If n is a goal node, exit with the solution obtained by tracing back the pointers from n to s
5. Otherwise expand n, generating all children and attach to them pointers back to n. For all children $\acute{n}$ of n:
   (a) If $\acute{n}$ is not already on OPEN or CLOSED, estimate $h(\acute{n})$ and calculate $f(\acute{n})$.
   (b) If $\acute{n}$ is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest $g(\acute{n})$.
   (c) If $\acute{n}$ required pointer adjustment and was found on CLOSED, place it on OPEN
6. Go to step 2

[Pearl,1985: 64-65].

3.5 *Heuristic Estimate of* $h(n)$

According to Felten, the selection of the proper heuristic function to estimate $h(n)$ is critical. A good heuristic allows the program to prune non-optimal branches of the search tree early in the search [Felten, 1988: 1501]. Kumar states that if $h(n)$ is very close to the actual cost, then most nodes expanded will be on the path to the optimal solution producing a very efficient algorithm [Kumar, 1990: 44].

Two of the most widely used admissible heuristics to generate $h(n)$ are the minimum spanning tree and the assignment problem. Both are polynomial time algorithms, but according to Kumar,

the assignment problem is one of the best heuristics for use with the TSP [Kumar, 1988: 124]. Therefore the heuristic function used in these algorithms is the assignment problem. Christofides defines the assignment problem as follows:

> Given a number of resources and a number of requesters of those resources, and the profit or usefulness of each resource to each requester in the form of a rating matrix where elements a_ij is the profit of assigning resource i to requester j, the problem is to assign each resource to one and only one requester in a way such that a given measure is optimized [Christofides, 1975: 287].

This definition implies the number of requesters and the number of resources are the same. In that case, the solution can be found in polynomial time. However, in cases where the number of requesters and resources are not equal can be solved by adding *dummy* resources or requesters to make the matrix square. The problem is now combinatoric in nature and in the class of NP-complete problems. When used in the TSP, the assignment problem has the same number of resources and requesters.

The assignment problem can be viewed as a matching of bipartite graphs. A bipartite graph is defined by Christofides as:

> a non-directed graph G = (X,A) is said to be bipartite if the set X of its vertices can be partitioned into two subsets X_a and X_b so that all arcs have terminal vertex in X_a and the other in X_b. A directed graph is said to be bipartite if its non-directed counterpart G' is bipartite [Christofides, 1975, 40].

Mathematically, the assignment problem can be stated as follows:

Given there are N requesters, W resources, and a cost matrix C

$$C = \|c_{ij}\|, c_{ij} \geq 0 \; for \; i = 1, 2, \ldots, N \; and \; j = 1, 2, \ldots, W \tag{3.5}$$

find an assignment matrix

$$X\|x_{ij}\| \tag{3.6}$$

such that

$$x_{ij} = \begin{cases} 1 \; if \; resource \; i \; is \; assigned \; to \; requester \; j \\ 0 \; otherwise \end{cases} \tag{3.7}$$

subject to the constraints

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} = minimum \sum_{i=1}^{N} x_{ij} = \sum_{j=1}^{W} x_{ij} = 1 \tag{3.8}$$

3.5.1 Assignment Problem Example As an example of the assignment problem, Figure 3.5 *A* is a 0/1 matrix representation of the requesters and the resources. A "1" in position x_{ij} of the matrix means resource j can be assigned to requester i and a "-" means it cannot be assigned. Another matrix, Figure 3.5 *B*, is constructed having the costs of each resource being assigned to each requester . After performing an element by element multiplication of the two matrices, the final 0/1 matrix showing the cost to perform each task is calculated and shown in Figure 3.5 *C*.

3.5.2 Assignment Problem Algorithm While there are many algorithms to solve the assignment problem, one of the most widely used is the *Hungarian Method* developed by Kuhn. This algorithm finds independent sets which have minimal (or maximal) costs. Bourgeois and Lassalle define a set of elements of a matrix to be independent if none of the elements are in the same row or column [Bourgeois and Lassalle, 1971: 14]. This restricts the allocation of one resource to one requester and vice versa. The independent sets are found by subtracting the smallest element of a row in the cost matrix from all other elements in the same row. Then the smallest element in each column in the cost matrix is subtracted from each element in that column. This results in every row and column having at least one null element for a N X N matrix. A row or column is *covered* when it contains only one null element. The smallest element in the uncovered rows/columns is then subtracted from all the uncovered elements and added to null elements of the covered rows/columns.

REQUESTERS	RESOURCES 0	1	2	3
0	1	1	-	1
1	-	1	-	-
2	-	1	1	-
3	1	1	-	1

A
AVAILABILITY MATRIX

REQUESTERS	RESOURCES 0	1	2	3
0	7	5	22	9
1	32	6	76	2
2	22	42	19	54
3	11	82	79	16

B
COST MATRIX

REQUESTERS	RESOURCES 0	1	2	3
0	7	5	INF	9
1	INF	6	INF	INF
2	INF	42	19	INF
3	11	82	INF	16

INF = INFINITY

C
FINAL MATRIX

Figure 3.5. Assignment Problem Cost Matrix

The process is repeated until all resources are covered by independent elements, in which case an optimal solution has been found [Kuhn, 1955: 25] .

The algorithm derived from the word description is:

1. Construct a cost matrix C_o, where each c_{ij} is the cost of the link in the bipartite graph between subgraphs $x_i \in X_a$ and $x_j \in X_b$.
2. Subtract the minimum element in each row of C_o from every element in that row.
3. Subtract the minimum element in each column of C_o from every element in that column.
4. For every row in the matrix with *only one* null element, mark the null element and cross out any other null element in that column.
5. If all rows are covered, i.e., contain a marked null element, then this corresponds to an optimal solution and exit the algorithm. If all rows are not covered, go to the next step.
6. For every column in the matrix with *only one* null element, mark the null element and cross out any other null element in that row.
7. If all columns are covered, exit with an optimal solution. If all columns are not covered, go to the next step.
8. Mark any row which does not contain any *marked* null elements.
9. Mark columns which have *unmarked* null elements in a marked row.
10. Mark the rows that have a marked null element in a marked column.
11. Repeat steps 9 and 10 until nothing else can be marked.
12. Cover all *unmarked* rows and all *marked* columns by drawing a line through them.
13. Find the minimum uncovered element and subtract it from all other uncovered elements and add it to the elements that are covered by both a row and column cover, i.e., the intersection of the lines.

14. Repeat steps 2 through 13 until an optimal solution is found. The cost of the optimal solution is found by summing the individual costs of the marked null elements in the original cost matrix C_o.

Using the cost matrix in Figure 3.6, an example assignment problem is solved. Notice in step 4 that the first null element to be marked was in the last row. Crossing out the two other nulls in the same column allowed the remaining null in the first row to be marked. Now the other null in the second row could be crossed out. Step 8 in Figure 3.7 begins the process of generating other possible combinations of null assignments. Step 13 in Figure 3.8 shows the new cost matrix after performing steps 8–13. With this new cost matrix, the algorithm is started again at step 2. Figure 3.9 step 5 shows the row and column null elements. The final solution is determined from assigning the null elements in a column (RESOURCE), to the row it is in (REQUESTER). In this example the solution is:

RESOURCE	REQUESTER
1	2
2	1
3	4
4	3

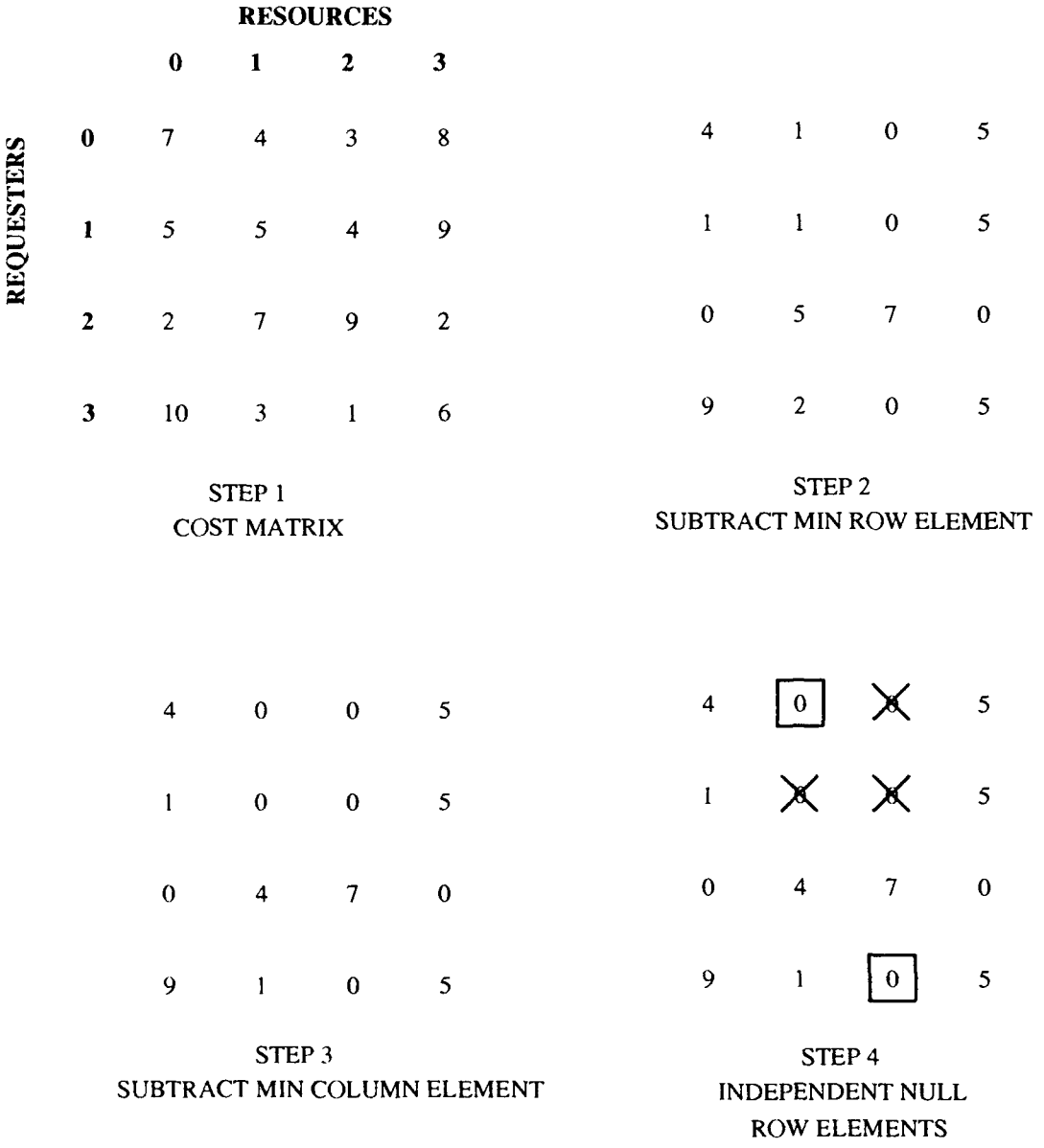
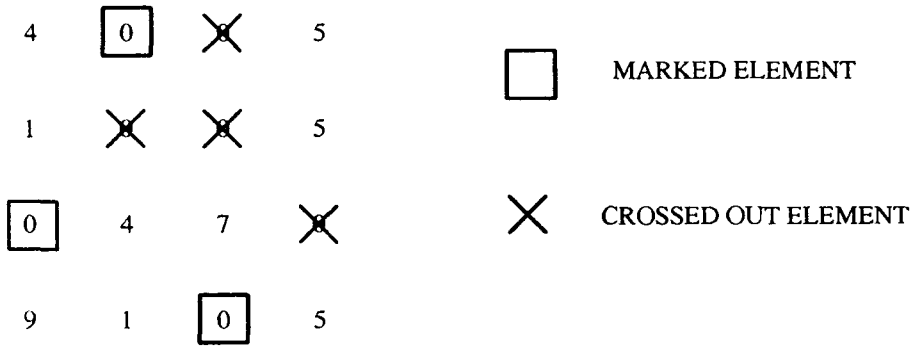


Figure 3.6. Assignment Problem Part 1



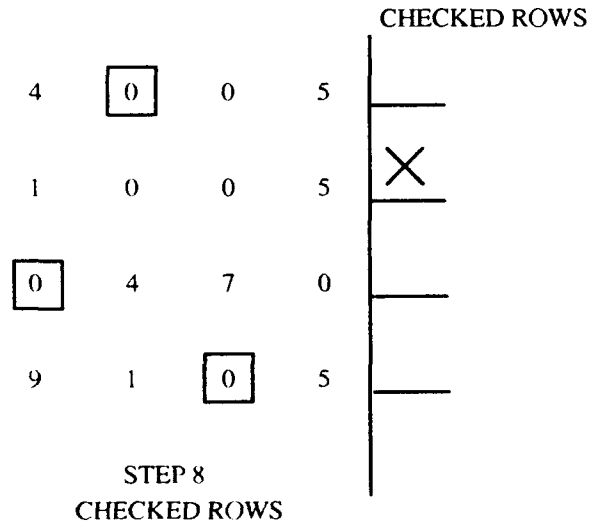
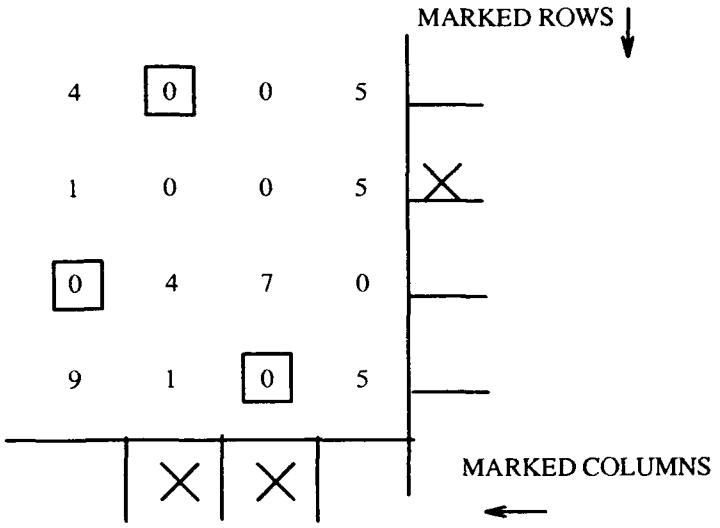
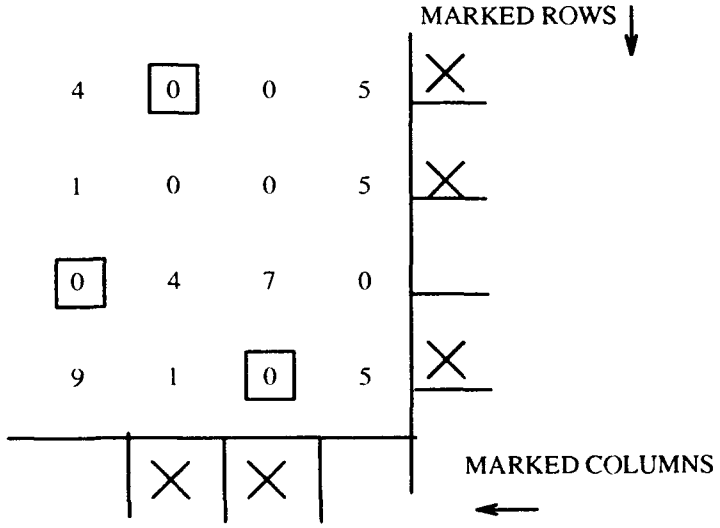


Figure 3.7. Assignment Problem Part 2

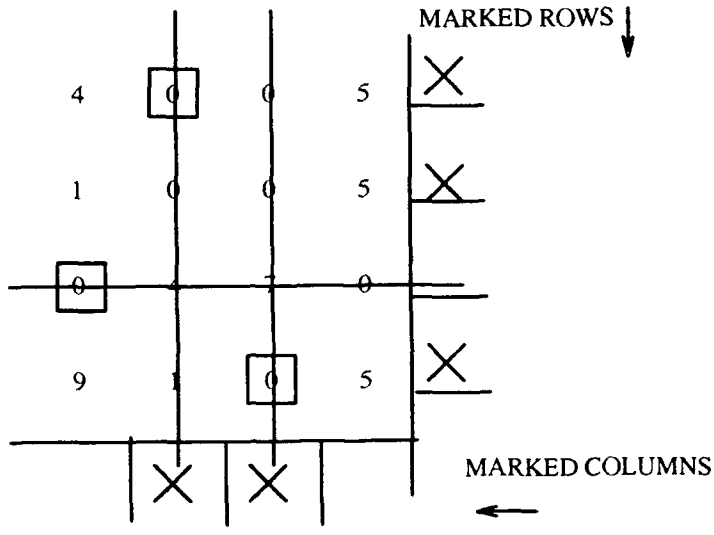


STEP 9
MARKED ROWS
AND COLUMNS



STEP 10
MARKED ROWS WITH
MARKED ELEMENTS IN
MARKEDF COLUMNS

Figure 3.8. Assignment Problem Part 3



STEP 12

3	0	0	4
0	0	0	4
0	5	8	0
8	1	0	4

STEP 13

NEW COST MATRIX

3	0	X	4
0	X	X	4
X	5	8	0
8	1	0	4

STEP 5

SOLUTION

Figure 3.9. Assignment Problem Part 4

3.6 High Level Design

Design of an algorithm can be divided into three broad categories, high level design, low level design, and implementation. High level design consists of the major steps required to perform the task and doesn't consider architectural peculiarities of the machine. Low level design adds more details to the algorithm and begins to customize it for a particular architecture. Implementation provides a finished program capable of running on a computer.

Designing a parallel algorithm has all the difficulty of designing a sequential algorithm with many other considerations besides. Parallel considerations include, mutual exclusion of data, control of the algorithm, timing between processors, load balancing, and decomposition techniques. Only control of the data and decomposition techniques are discussed in this chapter. The other considerations are discussed in Chapter IV during low level design.

3.6.1 Sequential TSP Algorithm

3.6.1.1 Terms and Definitions In all the algorithms discussed, there are some common terms and definitions which are given now. Chapter IV also provides more details of the data structures used and gives examples for some of the terms.

- num_cities - The number of cities to be visited in the problem
- NODE - A structure which has the fields vector, cost, and link. Vector contains the cities which have been visited, link points to the next NODE in the OPEN queue and cost is the cost of visiting the cities in vector
- BEST - A structure of type NODE which has the fields vector, cost, and link and contains the current best solution used to bound the search
- OPEN - The open list kept in an array of NODEs as described above. References can be made to any element of the queue and field of the NODE by using the element number and

the field name. For example, to compare the cost of the node on the front of the queue against the current BEST.cost, the following is used:

$$if(OPEN[q\_front].cost < BEST.cost)$$

- free list - A subset of the OPEN array which links elements of the array available to store NODEs
- WORK_REQUEST - Label used in the algorithm to identify messages sent by Workers to the Controller requesting a NODE from OPEN for expansion
- node_status - Status of Worker, either busy or available for work
- NEW_NODE -Label used in the algorithm to identify messages sending NODEs from the Workers to the Controller for insertion in the OPEN list
- DONE - Label used in the algorithm to identify the terminate message sent by the Controller to the Workers
- EXPAND_NODE - Label used in the algorithm to identify messages sending a NODE from the OPEN list on the Controller to a Worker

3.6.1.2 TSP ALGORITHM The basis for the parallel TSP search is the sequential TSP algorithm. The sequential algorithm was modified from the one developed by 1LT Mike Rottman to solve the TSP on the iPSC/1 hypercube [Rottman 1990: 97-105]. This algorithm uses the A* definition in Pearl [Pearl, 1988: 64] and the definition of TSP. The assignment problem is the function used to calculate the heuristic $h(n)$. The sequential algorithm is:

Sequential TSP
Build cost matrix
Perform depth first search of one node to obtain initial BEST.cost
Generate starting node in search tree

```
    Loop while (cost of NODE on front of OPEN ¡ BEST.cost)
        Remove NODE from OPEN
        Loop until all cities have been checked
            Add a city to end of partial tour of NODE removed from OPEN
            If city has not been visited in this partial tour
                Calculate h(n) and f for new partial tour
            If (new NODE.cost < BEST.cost) and (cities visited is a tour)
                BEST = NODE
            If (NODE.cost ¡ BEST.cost) and (cities visited is not a tour)
                Insert new NODE into OPEN
        END Loop until all cities have been checked
    END Loop while (cost of NODE on front of OPEN < BEST.cost
    Calculate/Collect results
END Sequential TSP
```

This algorithm removes a NODE from the front of OPEN and adds one city to the NODE.vector partial tour. It checks the resultant partial tour to see if the added city had already been visited and if the resultant tour was a complete tour. If the city was already in NODE.vector, the NODE is discarded. If the city was not in NODE.vector, was a complete tour, and at a lower cost than the current best cost, then the node becomes the new BEST. If it was not a tour and the cost was less than BEST.cost, the node was inserted into the OPEN queue for possible selection for expansion. This continues until all possible cities have been added to the original NODE.vector partial tour. The algorithm then removes the next NODE from OPEN and begins the cycle again. This continues until OPEN is empty or the cost of the NODE at the front of OPEN is greater than BEST.cost.

For example see Figure 3.10. The search begins by initially placing node 1 on OPEN. All possible children of node 1 are generated with their associated estimate cost to completion, f, and placed on OPEN. The NODE with the lowest f value is removed from OPEN and expanded. After each node is expanded, the children are placed on OPEN and the cycle is repeated. Notice that node 3 has the partial tour of 1 - 3 and each child of node 3 adds one city to that tour and generates a new cost. Notice also that node 7 only generates two children because the other possible cities have already been visited. The NODEs are kept in a priority queue so the *best* node is expanded

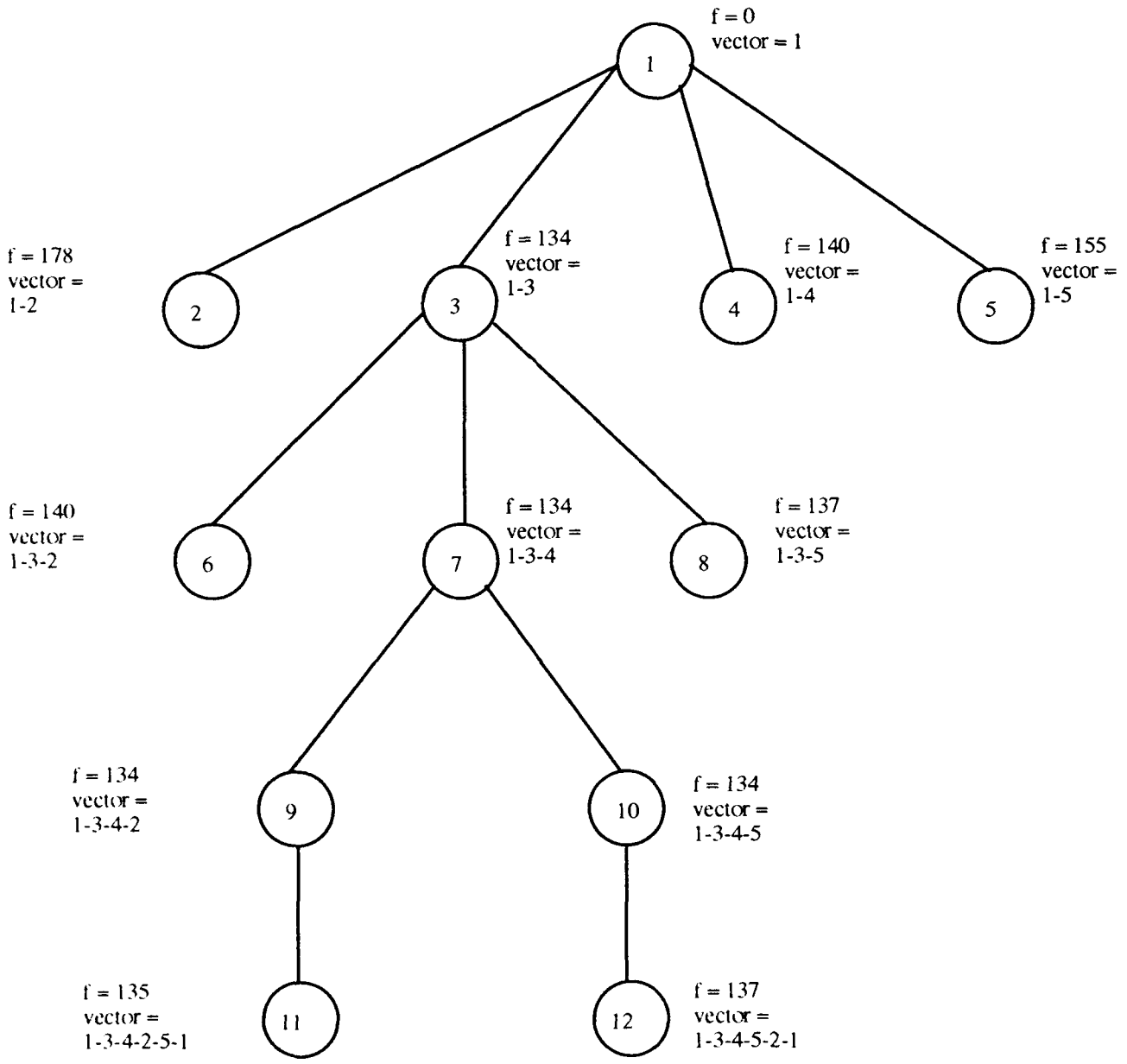


Figure 3.10. Example Search Graph for 5 Cities

next. The sorted OPEN list is the key element to making this a best first strategy and the function $f = g(n) + h(n)$ makes it A*.

3.6.2 Decomposition Techniques In determining the high level design, the first thing to consider is how the problem is to be decomposed onto the parallel computer. According to Ragsdale, two of the main decomposition techniques are data and control decomposition.

Data decomposition is where every processor has the same task and operates on different data sets. Information may or may not be passed between the processors as the programs are executed. Ragsdale provides three examples where data decomposition is well suited:

- Problems where the data is static. Examples include matrix operations or finite difference calculation on a mesh.
- Problems where the data structure is dynamic, but is somehow tied to a single entity. Examples include large multi-part problems with easily generated sub-problems.
- Problems where the domain is fixed, but the computation within the various regions of the domain is dynamic. For example, the search space of an NP-complete problem is bounded (no matter how large), but areas of the search graph can be dynamically generated for exploration

[Ragsdale, 1990: 4.4 - 4.5].

Control, or functional, decomposition focuses on the flow of control in an algorithm. Ragsdale lists two types of control decomposition. The first, functional decomposition, looks at a problem as a set of operations or functions. The functions are divided up and put on separate processors. Data which requires a particular function must be sent to the processor which has that function.

The other control decomposition technique is called *Worker/Manager*. One process is the "manager" and farms out tasks to the "worker" processors. The manager keeps track of the work to be done and assigns the work to the workers as they become idle [Ragsdale, 1990: 4.5].

For the first design, the Worker/Manager decomposition is used to control the overall flow of the algorithm and a data decomposition is used on the worker nodes. This allows the use of the centralized list for load balancing as discussed in Chapter II. Using data decomposition on the worker processors allows the large data sets to be manipulated without having to constantly pass information between processors. This allows different branches of the search tree to be explored simultaneously.

3.6.3 High Level Algorithms Ragsdale suggests the following general approach to designing parallel algorithms using data decomposition :

- Distribute the data
- Restrict the computation so that each processor updates its own data
- Put in the communication

[Ragsdale, 1990: 5.1].

Using the sequential algorithm as a starting point, two algorithms are developed. The first algorithm is the manager which distributes the data and controls the overall flow of the algorithm. The second algorithm is the worker and performs the actual computations required to execute the A* algorithm. The third step in the design process, communication, is accomplished in both the worker and manager algorithms. The designs developed are very similar to the sequential TSP design with the functions divided between the Control and Worker algorithms.

The high level Control design is:

High Level TSP Control Design
Receive cost matrix from host
Generate starting node in search tree
While (nodes still left on OPEN)
    Send work to idle Workers
End While (nodes still left on OPEN)

Terminate Workers
Collect results
END *High Level TSP Control Design*

The Control routines are very simple, well known types of routines and so little time will be spent discussing them. The Control algorithm contains most of the initialization and termination routines. The only Control routines that are not used in the sequential TSP algorithm are "send work to idle Workers" and "terminate Workers". These routines are peculiar to a parallel implementation of the algorithm and provide control information to the workers. On a serial computer, the control is provided by the sequential nature of the algorithm. Since the statements are executed sequentially, there is no conflict over which statement is executed next or when the program terminates.

The high level Worker design is:

High Level TSP Worker Design
Receive cost matrix from host
While (not terminated)
Request and receive work from Control
Perform A* search
Broadcast solution if better than current best solution
Send local OPEN list to CONTROL
Send results to Control
End While (not terminated)
END *High Level TSP Worker Design*

The heart of the Worker algorithm is the "Perform A* search" routine. This routine is the sequential version of the TSP A* algorithm executed on multiple processors. The routines which determine the efficiency and speedup of the parallel algorithm are "Request and receive work from Control", "Broadcast solution", and "Send local OPEN list". These routines, along with their corresponding routines on Control, control when and how information is passed between processors. These routines are further discussed in low level design.

3.7 Summary

This chapter presented the methodology used to attack the research, the metrics used to evaluate the efficiency of the algorithms developed, and the high level design. Also discussed was a more detailed explanation of the of the TSP problem. The preliminary design was partitioned into two separate algorithm based upon the "worker/manager" concept of control decomposition. The worker algorithm was further refined using data decomposition. The next chapter provides detailed design of the algorithms along with the data structures and functions used to implement the main algorithms.

IV. Low Level Design and Implementation

4.1 Introduction

In this chapter,the data structures used in the programs, the routines which comprise the programs, and the rationale behind the decisions to use each routine or structure is discussed. Diagrams showing the relationship between programs and routines is provided in Appendix A.

4.2 Data Structures

The basis for most of the data stored in these programs is the following C language structure:

```
typedef struct {
   int     vector[VECTOR_SIZE+1];
   int     cost;
   int     link;
} NODE;
```

The type *NODE* has three fields, each of which is of type integer. The first field, vector, is an array of size VECTOR_SIZE. VECTOR_SIZE equals the number of cities in the problem. The order of the cities in NODE.vector is the order in which the cities are visited. The second field in NODE, cost, contains the cost of the partial or complete tour stored in vector. The final field, link, is used as a pointer to the next NODE when NODEs are stored on the OPEN list in a queue.

VECTOR	COST	LINK
ARRAY OF INTEGERS, NUMBER OF ELEMENTS EQUALS NUMBER OF CITIES	INTEGER	INTEGER

Figure 4.1. Structure of Type NODE

Another data structure is the queue used to store nodes waiting to be expanded. The queue is an array of NODEs linked using the NODE.link field to determine which element in the array is

next on the queue. Also in the array is a list of elements which have no data and are considered *free* elements. Initially, all the elements are on the free list. Associated with the queue are pointers to the front of the queue and free lists with variables to show the queue status and queue length. NODEs are inserted into the queue using an insertion sort so that the NODE with the smallest cost is at the front of the queue. The free list is kept as a last-in-first-out queue.

An array was used to implement the OPEN queue for two reasons. The main reason is that the hypercube is a distributed memory architecture and each processor has its own distinct memory. This architecture does not allow information passing using a linked list since the pointers to memory locations have no meaning on a different processor. Therefore, if a linked list is used, the queue on each Worker must be put into an array before transmitting it to the centralized list kept on the Control processor. The other reason is that while the OPEN queue can become unmanageably large for A* search problems, the manner in which the heuristic estimate for h(n) is calculated is relatively accurate and keeps the OPEN list from becoming very large. This allows the array to be of a manageable size, around 9,000 elements, and still be large enough to contain the OPEN list.

In Figure 4.2, the array is comprised of 10 elements, six on the queue and four on the free list. The front of the queue is pointed to by q_front and the front of the free list by freeptr. The status is given by q_status as busy and the q_length is six.

After NEW_NODE is generated and inserted into the queue, the array and associated variables are as shown in Figure 4.3. Removing a NODE from the queue results in the configuration shown in Figure 4.4.

ELEMENT	VECTOR	COST	LINK
0	1-5-6	125	3
1	1-3-9-4-5-7-8-2	180	6
2	0-0-0-0-0-0-0-0-0-0	9999	8
3	1-2-3-4-6-9	139	1
4	0-0-0-0-0-0-0-0-0-0	9999	2
5	1-5-7-9-6-8-2	102	9
6	1-4-3-2	192	-1
7	0-0-0-0-0-0-0-0-0-0	9999	-1
8	0-0-0-0-0-0-0-0-0-0	9999	7
9	1-6-5-7-8-9-10	119	0

q_front = 5

freeptr = 4

q_status = BUSY

end of list = -1

q_length = 6

Figure 4.2. Structure of the OPEN Queue

ELEMENT	VECTOR	COST	LINK
0	1-5-6	125	3
1	1-3-9-4-5-7-8-2	180	6
2	0-0-0-0-0-0-0-0-0	9999	8
3	1-2-3-4-6-9	139	4
4	1-4-6-8-9-10-3-5	177	1
5	1-5-7-9-6-8-2	102	9
6	1-4-3-2	192	-1
7	0-0-0-0-0-0-0-0-0	9999	-1
8	0-0-0-0-0-0-0-0-0	9999	7
9	1-6-5-7-8-9-10	119	0

q_front = 5

freeptr = 2

q_status = BUSY

end of list = -1

q_length = 7

NEW_NODE	1-4-6-8-9-10-3-5	177	

Figure 4.3. Inserting into the OPEN Queue

ELEMENT	VECTOR	COST	LINK
0	1-5-6	125	3
1	1-3-9-4-5-7-8-2	180	6
2	0-0-0-0-0-0-0-0-0-0	9999	8
3	1-2-3-4-6-9	139	4
4	1-4-6-8-9-10-3-5	177	1
5	0-0-0-0-0-0-0-0-0-0	9999	2
6	1-4-3-2	192	-1
7	0-0-0-0-0-0-0-0-0-0	9999	-1
8	0-0-0-0-0-0-0-0-0-0	9999	7
9	1-6-5-7-8-9-10	119	0

q_front = 9

freeptr = 5

q_status = BUSY

end of list = -1

q_length = 6

Figure 4.4. Deleting from the OPEN Queue

	1	2	3	4	5
1	999	13	54	79	23
2	46	999	2	89	53
3	22	52	999	16	87
4	27	59	32	999	17
5	0	44	73	16	999

Figure 4.5. Cost Matrix Example

The other data structure is the matrix used to store the cost of traveling between cities. This is a square matrix with the row/column lengths equal to the number of cities in the problem. Each element of the matrix is an integer value. For example, in Figure 4.5 the cost of going from city 5 to city 3 is 73 while the cost of going from city 3 to city 5 is 87. Notice the cost matrix is not symmetric. Changes to the data structures are discussed later in the chapter as appropriate for algorithm changes.

4.3 Low Level Design

The low level design provides more details of the program and considers the architecture of the computer on which it will be running. Since these programs will run on the iPSC/2 hypercube, its message passing protocol and communication time must be considered. Also, the hypercube does not have an efficient interface between the user and processors. For this reason an additional algorithm, *Host*, is run on the host processor to provide the user interface.

This section also describes in greater detail the high level design developed in Chapter III. The Control program is discussed first, then the Worker program, and finally the functions used to implement specific actions of the programs. For each program, a pseudo code program is given.

4.3.1 Random City Generator A random number generator is used to build the cost matrix which is then stored in a file to be read into the main program later. This was done to allow repeated testing of the same problem using different size and possibly different types of hypercubes. The first element stored in the file is the number of cities in the problem. The random number generator assigns values of from 0–99 for the costs to travel between cities.

4.3.2 Control Program As developed in Chapter III, the Control high level design is as follows:

High Level TSP Control Design
Build cost matrix
Perform depth first search on one node to determine initial BEST.cost
Generate starting node in search tree
While nodes still left on OPEN or any Worker BUSY
    Send work to idle Workers
Terminate Workers
Collect results
END *High Level Control Design*

The high level Control design is further developed by adding communication requirements and more specific details to the algorithm and is shown below:

Low Level TSP Control Design
Receive cost matrix from host
Perform depth first search on one node to determine initial BEST.cost
Generate starting node in search tree
While ((OPEN not empty) or (any Worker busy)) loop
    While (WORK_REQUEST message from Workers) loop
        Receive *WORK_REQUEST message*
        Identify Worker which sent message
        Set appropriate node_status to available
    End While (WORK_REQUEST from Workers) loop
    While ((NEW_NODE message from Workers) and (OPEN not full)) loop
        Receive NEW_NODE message from Worker
        Insert NODE into OPEN
    End While ((NEW_NODE from Workers) and (OPEN not full))
    While ((OPEN not empty) and (Worker is available) and
            (front NODE on OPEN cost < best.cost)) loop
        Delete NODE from front of OPEN
        Send NODE to Worker for expansion
    End While ((OPEN not empty) and (Worker is available))
    If (NEW_BEST message from Worker)
        Receive NEW_BEST message from Worker
        If (NEW_BEST.cost < current best.cost)
            current best = NEW_BEST
            Prune OPEN list of NODEs with costs $\geq$ best.cost
        End If (NEW_BEST.cost < current best.cost)
    End If (NEW_BEST from Worker)
End While ((OPEN not empty) or (any Worker busy)) loop
Send DONE message to all Workers
Collect results from Workers
END *Low Level Control Design*

As long as the OPEN list is empty or the Workers are not all idle, the Control processor continually polls the receive buffers of the iPSC/2 for a message from the Worker processors and takes appropriate action when a message is received. For example, if a NEW_BEST message is received, the cost is compared to the current BEST.cost and then the OPEN list is pruned of unnecessary NODES. When the while loop is exited, a terminate message is sent to all Workers. Finally, data is collected from the Workers.

If more than one Worker is available to send work to, the algorithm selects the processor with the lowest number. For example, if processors 3,4,and 5 are all available, processor 3 and then processor 4 receive work. If processor 3 requests work again before processor 5 receives work, it will receive the work before processor 5. This skews the efficiency of the individual processors so lower number processors have higher efficiencies.

Terminating a centralized list A* algorithm requires that the Worker processors be idle and the OPEN list be empty. Idle Workers are determined by the *Worker Busy* variable. When work is sent to a Worker, busy is set to "true", and set "false" when a work request is received by the Control processor. The OPEN list is checked at the beginning of each loop through the algorithm to ensure it has valid work. If either the OPEN list is not empty or any Worker processor is not idle, the algorithm continues.

4.3.3 Worker Program The high level design from Chapter 3 for the Worker is:

High Level TSP Worker Design
  Build cost matrix
  While (not terminated)
    Request and receive work from Control
    Perform A* search
    Broadcast solution if better than current best solution
    Send local OPEN list to CONTROL
    Send results to Control
  End While (not terminated)
END *High Level TSP Worker Design*

Like the Control algorithm, the Worker algorithm is further developed in the low level design.

Low Level TSP Worker Design
  Receive cost matrix
  While (not terminated by DONE message)
    Send WORK_REQUEST message to Controller
    Receive EXPAND_NODE message from Controller
    Loop until all cities have been checked
      Add a city to end of cities which have been visited
      If new city has not been visited in this partial tour
        Calculate the cost ($h(n)$ and $f(n)$) for new partial tour
        If (new NODE.cost < BEST.cost) and (NODE.vector is a tour)
          BEST = NODE
          Broadcast NEW_BEST to all processors
        If (NODE.cost < BEST.cost) and (NODE.vector is not a tour)
          Insert new NODE into OPEN
    END (Loop until all cities have been checked)
  END (Loop while (not terminated by DONE message))
  Send results to Controller or Host
END *Low Level TSP Worker Design*

This algorithm is very similar to the sequential TSP developed in Chapter 3. See Figure 3.10 for the detailed explanation and for an example of this algorithm. The main differences from the sequential TSP algorithm are the communication between processors and the algorithm control is provided by the Control algorithm.

4.3.4 Host Program An algorithm to provide interface with the user is provided by the *Host* program which runs on the iPSC/2 system resource manager (SRM), or host processor. This algorithm prompts the user for information needed to run the TSP program, performs initialization, loads the Control and Worker processors, and displays final results. The design for this algorithm is:

Low Level TSP Host Design
  Print initial messages
  Request and receive file name where cost matrix is stored
  Copy cost matrix
  Load Control and Worker programs

```
   Send cost matrix to Control and Worker programs
   Receive results from Control and Worker programs
   Print results and termination messages
END Low Level TSP Host Design
```

4.3.5 Subroutines Much of the work in the Control and Worker main programs is performed using calls to subroutines or functions. This section discusses the functions used in both Control and Worker programs.

The most important function is the assignment problem used to calculate $h(n)$. This algorithm is discussed in detail in chapter III. Another function, *Tour*, traces through each city of NODE.vector to see if the path goes through each city only once and ends at city 1. It returns a boolean flag stating whether NODE.vector is a tour. The algorithm is:

```
Tour
   Initialize test array of size num_cities to 0
   Set Tour flag to FALSE
   Mark city 1 as visited in test array
   While (city not visited twice) loop
       Go to next city in NODE.vector
       Mark city as visited in test array
   End While (city not visited twice)
   If (all cities visited once) and (end at city 1)
       Set Tour to TRUE
   End If (all cities visited once) and (end at city 1)
END Tour
```

The function to determine if the city added to the end of NODE.vector is a feasible selection is *in_path*. This function traces through NODE.vector to determine if the city has already been visited in this partial tour. It returns a boolean flag stating whether the city has already been visited. The algorithm is :

```
In_path
   Set In_path flag to FALSE
```

```
While (cities not visited in NODE.vector) loop
    If (city in NODE.vector = city being added)
        Set In_path flag to TRUE
    End If (city in NODE.vector = city being added)
End While (cities unvisited in NODE.vector)
```

Because the C language does not support direct copying of arrays, a function called *Copy_node* was made. This function copies the array stored in NODE.vector to another variable of type NODE. The other fields in the NODE structure are also explicitly copied. The algorithm is:

```
Copy_node
    While (unvisited elements in NODE_1.vector) loop
        NODE_2.vector = NODE_1.vector
    End While (unvisited elements in NODE_1.vector)
    NODE_2.cost = NODE_1.cost
END Copy_node
```

Unlike a sequential algorithm, several processors could locate solutions which are better than the current best solution and broadcast it at relatively the same time. The broadcasted message is 450 bytes and according to Bomans and Roose, this size message takes approximately 800 μ seconds to transmit. Because of the near simultaneous broadcasts and communication delays, a processor could receive a NEW_BEST message which was higher than the current BEST.cost stored at that processor [Bomans and Roose, 1989: 16]. To insure the best solution is stored in BEST, every NEW_BEST message is compared against BEST.cost and the smaller value is returned. The algorithm to perform this is *Get_best* and is:

```
Get_best
    While (receiving NEW_BEST messages) loop
        If (NEW_BEST.cost < BEST.cost)
            BEST = NEW_BEST
        End If (NEW_BEST.cost < BEST.cost)
    End While (NEW_BEST message)
END Get_best
```

The OPEN list is stored in an array of NODEs. Four functions control all actions associated with data manipulation of the OPEN queue. The first function, *q_init*, initializes all elements in the array including the NODE.vector array within each element. It also initializes the link and cost fields of NODE in each element of OPEN. Finally, all other variables associated with the OPEN queue are initialized. The algorithm for *q_init* is:

```
q_init
   For (all the elements on OPEN)
        For (all the elements in NODE.vector)
            Set NODE.vector field to 0
        End For (all the elements in NODE.vector)
        Cost = INFINITY
        Link = next element of OPEN
   End For (all the elements on OPEN)
   Set link field of last element on OPEN to (end of file marker)
   q_status = EMPTY
   q_length = 0
   Pointer to free list = 0
   Pointer to OPEN list = (end of file marker)
   End (all the elements on OPEN)
END q_init
```

The NODEs are deleted from OPEN by the algorithm *delete_q*. This algorithm deletes the NODEs, changes the queue length variable, and adjusts the pointers to the front of the OPEN queue and free list. Error checking is also performed to provide a warning message if the algorithm attempts to delete a NODE from an empty queue. Finally, the status of the queue is checked and changed as needed. The algorithm is:

```
delete_q
   If the queue is empty print warning message
   If the queue is not empty
        Decrement queue length by 1
        Remove NODE from the front of the queue
        Adjust pointer to the front of the queue to point to the new front
        Adjust pointer to the front of the free list to point to the new front
        Adjust the queue status as appropriate
   end (If the queue is not empty)
```

END *delete_q*

The third function which manipulates the queue is *insert_priority*. This algorithm performs an insertion sort of the NODEs into the OPEN queue, changes the queue length variable, and adjusts the pointers to the front of the OPEN queue and free list. This algorithm also performs error checking and updates the queue status. The algorithm is:

insert_priority
  If the queue is full print warning message
  If the queue is not full
    Increment queue length by 1
    Insertion sort the NODE into the OPEN queue
    Adjust pointer to the front of the queue to point to the new front
    Adjust pointer to the front of the free list to point to the new front
    Adjust the queue status as appropriate
  end (If the queue is not full)
END *insert_priority*

The final function to work with the queue is *prune_q*. After a NEW_BEST solution is found, this function is used to bound the search by removing, or pruning, states from the search space tree. This is done by removing from OPEN all NODEs which have a cost greater than or equal to the cost of the new solution. Notice that NODEs with equal cost are also eliminated since the algorithm only searches for a best solution not all best solutions. The algorithm is:

prune_q
  Traverse the OPEN queue until BEST.cost *geq* NODE.cost
  Delete all NODEs in OPEN beyond the current NODE
  Adjust pointers in the free list
  Adjust the queue status as appropriate
END *prune_q*

4.4 Distributed List

Quinn theoretically proved and Abdelrahman and Mudge demonstrated that as the number of processors increased, the communication to/from the master processor allocating work to slave processors eventually becomes a bottleneck. To eliminate this bottleneck, distributed list algorithms are used [Quinn, 1990: 385] [Abdelrahman and Mudge, 1988:1496-1498].

In this section, the modifications required to change the parallel TSP control from a centralized list to a distributed list (DL) are discussed. This entails changing the high level decision of using the functional worker/manager decomposition and use only data decomposition on all processors. This eliminates the Control processor, but adds another Worker processor. The global OPEN queue is now maintained in a local OPEN queue on each processor. Extra communication between processors is required to perform the tasks previously done by the Control processor such as load balancing and program termination. This section looks at implementing distributed list queues with and without load balancing.

4.4.1 DL Without Load Balancing There are two main methods used to implement a distributed list queue. The first method generates and distributes an initial work load and then no load balancing, or work sharing, is performed. Work which is generated by a processor stays on that processor. When a processor finishes its work, it remains idle until all processors finish. If the work generated by the processors is not approximately equal, processors may be idle for a relatively long period of time waiting for all the processors to finish [Ma and others, 1988: 1509-1511].

The new design for distributed list TSP without load balancing is:

TSP Worker Without Load Balancing Design
Receive cost matrix from host
Perform depth first search on one node to determine initial BEST_cost
Generate starting node in search tree
Distribute descendents of initial node among all processors
While (not terminated by DONE message)
While (OPEN not EMPTY)

```
    Remove NODE from OPEN queue
    Loop until all cities have been checked
        Add a city to end of cities in NODE.vector which have been visited
        If new city has not been visited in this partial tour
            Calculate the cost (h(n) and f) for new partial tour
            If (new NODE.cost < BEST.cost) and (NODE.vector is a tour)
                BEST = NODE
                Broadcast NEW_BEST to all processors
            If (NODE.cost < BEST.cost) and (NODE.vector is not a tour)
                Insert new NODE into OPEN
    END (Loop until all cities have been checked)
    END While (OPEN not EMPTY)
    Terminate the processors
  END Loop while (not terminated by DONE message)
  Send results to Host
END TSP Worker without Load Balancing Design
```

Again, this is just the sequential TSP algorithm with communication and control to allow the parallel operation. After the children of the initial node are generated, they are distributed equally among the processors. Each processor then performs the sequential TSP until all of the NODEs on its local OPEN have been explored.

To terminate the parallel program, all processors must be idle. This is determined by sending a message, RING, which is only received when the processor is idle. The body of the RING message is empty and just the fact that the message was received is significant. The processor identified by the hypercube operating system as *node 0* initiates RING when its OPEN queue is empty and sends it to the next logically numbered processor. Once a processor is idle, it receives the RING and passes it to the next processor. When processor 0 receives the RING again, all processors are idle and a DONE message can be sent terminating all the processors.

4.4.2 DL With Load Balancing The main deficiency of the DL without load balancing algorithm is the that work could be unevenly distributed and some processors are idle will others still have large amounts of work to perform. As Ma and others show, the efficiency of distributed lists *without load balancing can be very low* [Ma and others, 1988: 1509-1511]. The obvious solution is to have an idle processor request work from a busy processor. The idle processor first requests

work from its nearest neighbors and then request work from all other processors one at a time. Once an idle processor receives work, no more requests for work are issued. The busy processors must periodically check its receive buffers to see if it has received a work request. The only way a processor can become idle is if *all* processors are either idle or do not have enough work to share.

Felten, Ma, Penky and Miller, Cvetanovic and Nofsinger, and many others discuss when it is appropriate for a processor to share work. They all agree there is a trade off between sharing the work to keep a processor from being idle, the communication overhead involved with the work sharing, and the possibility of a processor sharing too much work and having to immediately request work itself. They state the measure of when to share, β, is problem specific and must be determined experimentally. [Felten 1988, 1504] [Ma and others, 1988: 1507] [Miller and Penky, 1989: 133] [Cvetanovic and Nofsinger, 1990: 87] . Since nothing was found in the literature to provide any guidance to the factors which influence β, this research investigates these factors.

According to Felten, there are two requirements to terminate a *DL* process with load balancing on a hypercube. The first requirement is the same as with the DL without load balancing that all processors be idle. The second is that all messages have been received [Felten, 1988: 1502]. The requirement to have received all messages insures no work was distributed to a processor but not received by that processor. To keep track of the number of messages sent, each processor keeps a local count of the messages sent/received. A variable is incremented when a message is sent and decremented when a message is received. To terminate the process, the local message count can be any number, but the global message count *must* equal zero. To accomplish this, the Ring message body from the DL without load balancing is changed to a data structure of the form:

```
typedef struct {
   int     message_count;
   int     num_done;
} FINISHED;
```

This termination algorithm differs from Felten's termination in the method used to determine if all processors are idle. Like the DL without load balancing algorithm, the DL with load balancing algorithm uses processor 0 to again initiate the RING when it becomes idle and send it to the next logically numbered processor. If a busy processor receives the RING, it just sends it to processor 0 and continues working. When an idle processor receives the ring, it increments the number done and adds its message count to the total message count. When processor 0 receives the RING, it checks FINISHED to validate that all processor are finished and all messages received. If either condition is not met, processor 0 again sends the Ring.

Felten's algorithm differs by having all processors, busy or idle, send the RING to the next processor. When processor 0 receives the RING, it checks to see if any processor is busy. Again, if either condition is not met, processor 0 again sends the Ring. In the DL with load balancing algorithm, at most one busy processor is interrupted by the RING message. In Felten's algorithm, at best only one busy processor is interrupted and at worse $[(numberofnodes) - 1]$ busy processors are interrupted [Felten, 1988: 1503]

Beard modified Felten's algorithm for termination and also used it for load balancing. A busy processor handles the RING similar to the Felten algorithm and sends it to the next processor. However, when an idle processor receives the RING, it differs from Felten's algorithm in that the RING now allows the processor to request work. Since only one processor can have the RING at a time, only one processor can be requesting work. Especially on computers with a large number of processors, many processors could be idle waiting to request work from busy processors [Beard, 1990: 4.25-4.28]. As implemented in this research, idle processors immediately and independently request work from their neighbors.

Beard also implemented the RING algorithm as a separate process on each processor and used context switching between the search algorithm and the RING algorithm [Beard, 1990: 4.25-4.28]. This appears to be an inefficient way to perform termination and load balancing. Both of these

tasks occur only at very specific points in the algorithm and are easily controlled. Also, context switching requires calls to the operating system which stop and start the different algorithms. All this incurs overhead not required if the functions are called from the main algorithm without context switching.

The DL with load balancing is:

```
TSP Worker With Load Balancing Design
   Receive cost matrix from host
   Perform depth first search on one node to determine initial BEST.cost
   Generate starting node in search tree
   Distribute children of initial node among all processors
   While (not terminated by DONE message)
        While (OPEN not EMPTY)
        Remove NODE from OPEN queue
        Check for RING
        Check for WORK_REQUEST message from another processor
        Loop until all cities have been checked
             Add a city to end of cities in NODE.vector which have been visited
             If new city has not been visited in this partial tour
                  Calculate the cost (h(n) and f) for new partial tour
                  If (new NODE.cost < BEST.cost) and (NODE.vector is a tour)
                       BEST = NODE
                       Broadcast NEW_BEST to all processors
                  If (NODE.cost < BEST.cost) and (NODE.vector is not a tour)
                       Insert new NODE into OPEN
        END (Loop until all cities have been checked)
        END While (OPEN not EMPTY)
        Send WORK_REQUEST to other processors
        Terminate the processors
   END Loop while (not terminated by DONE message)
   Send results to Host
END TSP Worker With Load Balancing Design
```

The algorithm to terminate the process is as follows:

```
Terminate
   If (my node number is 0)
        Initialize the RING message
        While (not all processors are idle) or (not all messages received)
             Check for WORK_REQUEST message from another processor
             Send the RING to node 1
```

```
            Receive the RING
        END while (not all processors are idle) or (not all messages received)
                Send the DONE message
    END If (my node number is 0)
    If (my node number is not 0)
        Receive the RING
        Modify the num_done and message_count fields to FINISHED
        Send RING to next processor
        Wait for DONE message
    END If (my node number is not)0
END Terminate
```

The other functions discussed pertain to sharing work between processors. The first function, send_work_request, sends a WORK_REQUEST message to other processors and waits for their response. This function also receives any work sent by another processor in response to this message.

The algorithm is:

```
send_work_request
Send WORK_REQUEST message to nearest neighbors
If (neighbor has work)
    Receive work from neighbor
    Insert work into OPEN list
    Expand the NODEs
END If (neighbor has work)
If (neighbor has no work)
    Send WORK_REQUEST message to all other processors
        If (processor has work)
            Receive work from processor
            Insert work into OPEN list
            Expand the NODEs
        END If (processor has work)
END If (neighbor has no work)
END send_work_request
```

The last function, share_work, is activated when a WORK_REQUEST message is received. It determines if the processor has enough work to share, transmits with either a TRUE or FALSE response to the requesting processor, and sends the work if appropriate.

If λ is the parameter which determines if there is enough work to share, Jansen and Sijstermans state that λ should be chosen in such a way to keep all the processors busy, but not let communication overhead dominate the process. If λ is too small, a processor could share work and then immediately have to request work itself. If λ is too large, processors could be idle while other processors have a relatively large OPEN list [Jansen and Sijstermans, 1989: 273].

How many NODEs to share is determined in two ways. First, if the OPEN list is larger than 20, then 10 NODEs are sent to the requesting processor. Requiring the OPEN list to be larger than 20 developed from experience in running the algorithm. If the OPEN list is less than 20 but larger than a predetermined value, then half of the OPEN list is sent. If the OPEN list is smaller than the predetermined value, no work is shared. The number used to determine how much work to share and the predetermined value, β, are problem specific [Cvetanovic and Nofsinger, 1990: 87].

Again, one goal of this research is to provide guidelines when setting these values.

The algorithm is:

```
share_work
   Receive the WORK_REQUEST message
   If (there is enough work to share)
        Send work to requesting processor
   If (there is not enough work to share)
        Send FALSE response to requesting processor
END share_work
```

4.5 A* Variations

As discussed in Chapter II, this research investigates two variations on the A* algorithm. This section provides a more detailed discussion of the algorithms and gives the algorithms.

*4.5.1 IDA** As the example in Chapter II shows, the main difference between A* and IDA* is IDA* performs a limited depth first search on the node selected for expansion by the A* portion of the algorithm.

To implement the parallel IDA* algorithm, changes were made to the centralized list Worker algorithm. The first change is to build another queue using the same structures and functions as for OPEN, but used to store the NODEs generated during the depth first search portion of the algorithm. The new queue is called *ida_q* and all the ida_q queue functions are named using the same name as the OPEN queue functions but add ida_ to the front.

The other change to the CL Worker algorithm is to add the control for the depth first search to the main algorithm. As long as child nodes do not exceed the cost of the original parent node, they are kept at that processor for expansion. Child nodes which exceed the cost of the parent are sent back to the Control processor for insertion into the OPEN list.

This differs from Korf's method of discarding all generated information except the threshold cost for the next iteration. This was done for two reasons. First, the algorithm did not have the problem with running out of memory that Korf experienced. The second reason was this research hoped to compare the IDA* and the continuous diffusion algorithms. Since neither algorithm is optimized in terms of execution time , only the number of nodes expanded would be used as a metric for comparison.

The IDA* Control and Host algorithms are the same as the CL Control and Host. The IDA* Worker algorithm is:

```
Low Level IDA* Worker Design
Receive cost matrix
While (not terminated by DONE message)
   Send WORK_REQUEST message to Controller
   Receive EXPAND_NODE message from Controller
   Insert received E_NODE into ida_q
   While (ida_q not empty) loop
      Expand node on front of ida_q
```

```
        Loop until all cities have been checked
            Add a city to end of cities which have been visited
            If new city has not been visited in this partial tour
                Calculate the cost (h(n) and f) for new partial tour
            If (new NODE.cost < BEST.cost) and (NODE.vector is a tour)
                BEST = NODE
                Broadcast NEW_BEST to all processors
            If (NODE.cost < BEST.cost) and (NODE.vector is not a tour) and (NODE.cost ≤ E_NODE.cost)
                Insert new node into ida_q
            If (NODE.cost < BEST.cost) and (NODE.vector is not a tour)
                Insert new NODE into OPEN
        END Loop until all cities have been checked
    END While (ida_q not empty) END Loop while (not terminated by DONE message)
Send results to Controller or Host
```
END*IDA* Worker Design*

4.5.2 TSP with Levels In trying to compare the IDA* algorithm against the centralized list or distributed list, the IDA* algorithm is at a disadvantage because of the assignment problem used to calculate the estimated cost to completion f(n). The advantage IDA* has is its ability to perform depth first searches once a node has been sent to the processor for expansion. Using the assignment problem to calculate f(n) also provides some depth first search, thus negating any advantage IDA* had.

To balance out the advantage provided by the assignment problem, the CL and DL algorithms were changed to force all solutions to be at the same level in the search graph. For example, in a 10 city TSP the solution must have all 10 cities. Normally, only 1 city is added at each level of the search graph. However, the assignment problem can, in some cases, provide a solution from any level in the graph. To counteract this, another variable was added to the NODE structure telling what level in the graph the state is. The new structure is :

```
typedef struct {
    int     vector[VECTOR_SIZE+1];
    int     cost;
    int     link;
    int     level;
} NODE;
```

If a solution is found but the level did not equal the number of cities, two things happened. First, the NODE.cost becomes the new global BEST.cost. Second, the variable which counts the number of NODEs expanded is incremented until the NODE.level equals the number of cities. This way the solution is always found at the lowest level of the search graph. Now at least the number of NODEs expanded can be compared between the IDA* and the other algorithms.

In the CL algorithm, only the Worker algorithm is modified to implement the use of the levels. The Control algorithm still only checks to see if the workers are idle and the OPEN list is empty before terminating the task. The DL algorithms and the IDA* algorithm have basically the same change as the CL Worker algorithm, so only the CL Worker is shown.

```
TSP Worker with Levels Design
   Receive cost matrix
   While (not terminated by DONE message)
        Send WORK_REQUEST message to Controller
        Receive EXPAND_NODE message from Controller
        Loop until all cities have been checked
             Add a city to end of cities which have been visited
             If new city has not been visited in this partial tour
                  Calculate the cost (h(n) and f(n)) for new partial tour
                  If (new NODE.cost < BEST.cost) and (NODE.vector is a tour)
                       Increment NODE.level to number of cities
                       BEST = NODE
                       Broadcast NEW_BEST to all processors
                  If (NODE.cost < BEST.cost) and (NODE.vector is not a tour)
                       Insert new NODE into OPEN
        END (Loop until all cities have been checked)
   END (Loop while (not terminated by DONE message))
   Send results to Controller or Host
END TSP Worker with Levels Design
```

4.5.3 Distributed List with Load Balancing and NODE Distribution Assuming that the sequential algorithm expands the fewest nodes, what is the best way to have parallel algorithms emulate the same ordering of nodes to be expanded. The centralized list algorithm very closely emulates the sequential algorithm, but its efficiency is limited when scaled to a large number of processors.

While the distributed list with load balancing insured all processors are kept busy until all the nodes have been examined, it does not mean they are all doing productive work. Saletore defines *wasted work* as work performed that to the right of the solution in the state space. For example see Figure 3-2. He assumes a left to right search of the state space. If the solution is state 1, then any node expanded to the right of state 1 is wasted work [Saletore, 1991: 4]. Felten describes *redundant work* as expanded nodes which the sequential algorithm would have eventually pruned off [Felten, 1988:1503].

PROCESSOR →

NODE COST ↓

0	1	2	3
124	123	155	178
124	123	156	178
126	124	163	189
128	125	166	192
129	125	167	192
130	127	167	193
130	127	170	193

Figure 4.6. Initial OPEN Lists

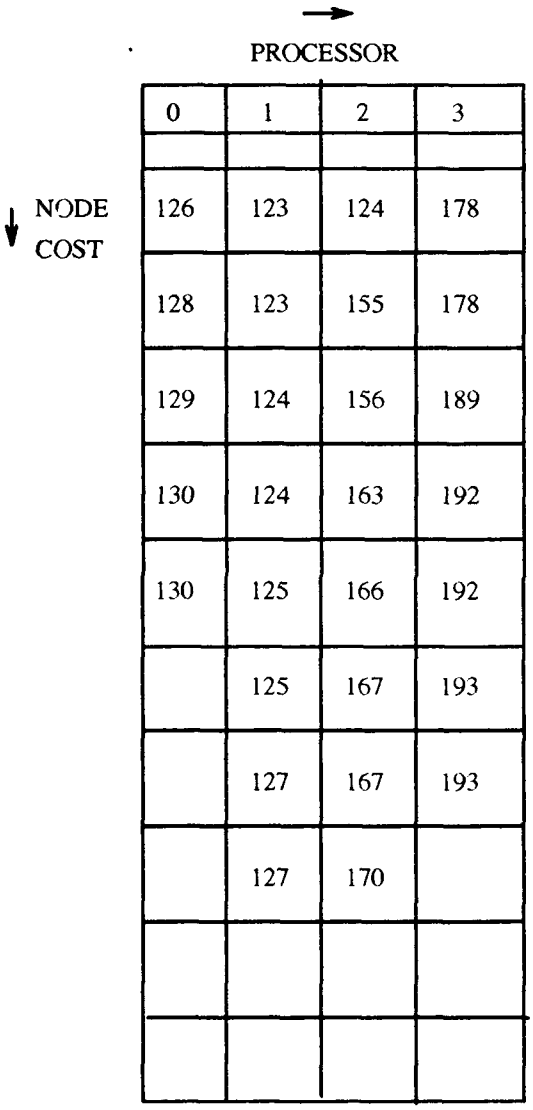


Figure 4.7. After Node 0 Distributed

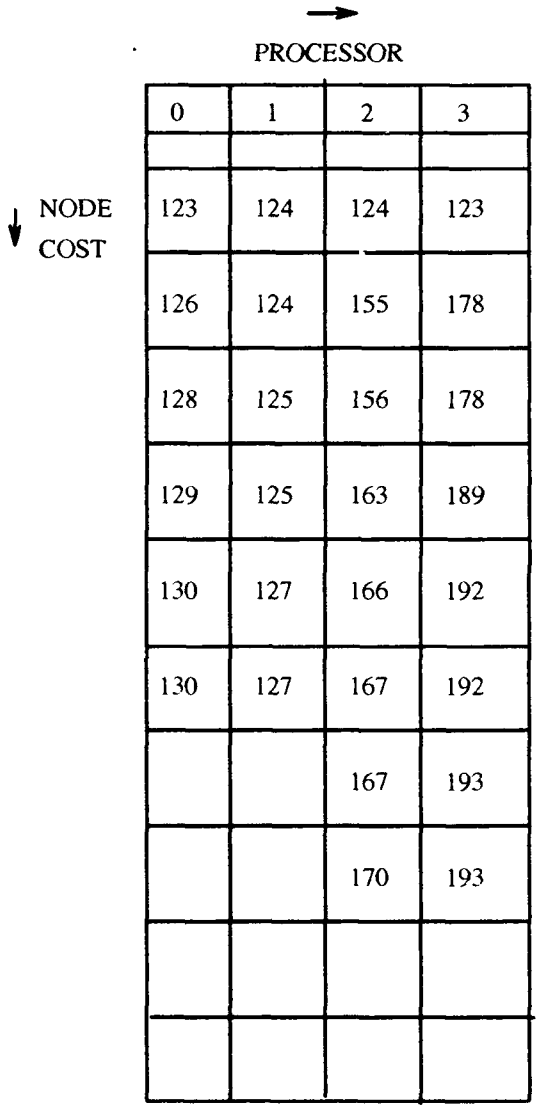


Figure 4.8. After Node 1 Distributed

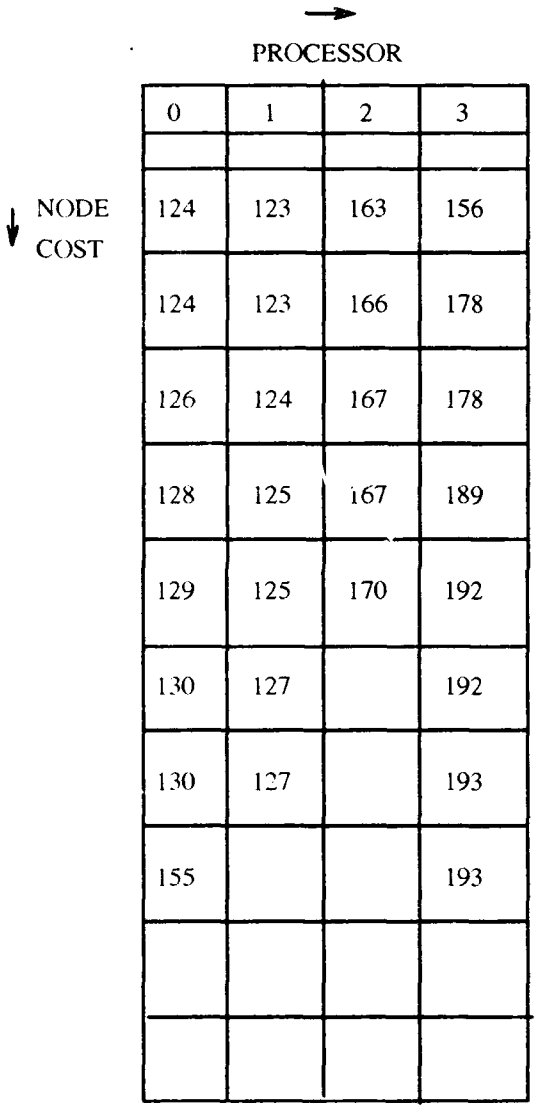


Figure 4.9. After Node 2 Distributed

In the sequential or CL algorithms, the NODEs would be ordered in non-decreasing cost so the lowest cost NODE would always be expanded next. However, with DL algorithms each processor has its own local list of nodes to expand and a processor could be expanding a node with a much higher cost than nodes on its neighbors. For example, Figure 4.6 shows the state of the OPEN lists at a certain point in time. Processors 0 and 1 have approximately equal cost NODEs at the head of the list while processors 2 and 3 have much higher cost NODEs. Obviously, we want the lowest cost nodes to be expanded next.

As discussed in Chapter II, Cvetanovic and Nofsinger propose a method they called *Continuous Diffusion* to evenly distribute the lowest cost NODEs. This algorithm periodically distributes a predetermined number of NODEs from the front of a processor's OPEN list to its nearest neighbors. Using the initial state of Figure 4.6, Figure 4.7 shows the state of the OPEN lists after processor 0 distributed 1 NODE to its nearest neighbors, processors 1 and 2. Figure 4.8 shows the state of the OPEN lists after processor 1 distributes NODEs. Notice how the front of the OPEN lists are much more uniform in cost. Figure 4.9 shows that if a processor with high NODE costs distributes, the NODEs sent are inserted farther down in the OPEN list [Cvetanovic and Nofsinger, 1990: 86-90].

Instead of distributing to its nearest neighbor, Felten suggests randomly distributing the NODEs [Felten, 1988, 1502]. One problem with this approach is there is no systematic way to distribute the lowest cost NODEs to other processors. A processor could never receive distributed NODEs and have much higher cost NODEs on its OPEN list.

One problem with these approaches is determining how often the processors should distribute NODEs. Cvetanovic and Nofsinger define δ as the number of nodes a processor expands before distributing from its OPEN list. The conflicting goals of minimizing extra search and load imbalance versus communication overhead determine the optimal value for δ [Cvetanovic and Nofsinger, 1990: 86-90]. Determining guidelines for setting the value of δ is one goal of this research.

Another problem with distributing NODEs is that a large number of NODEs with the same cost are generated. For example, on one run using 8 processors and 100 cities, every processor had *at least* 1000 NODEs with a cost of 134 at the front of the OPEN list. In this situation, it makes no sense to distribute if all the processors have NODEs with the same cost on OPEN.

To implement the distributed list with load balancing and NODE distribution algorithm, the distributed list with load balancing algorithm is modified to distribute NODEs. Additional control is added to the main algorithm to determine when to distribute. Also two more functions are provided to perform the actual distribution of the NODEs.

The first function, *distribute*, determines if there is enough work to distribute. This function has the same concerns about load balancing and communication overhead as the share_work function. Therefore the same guidelines used for share_work are used in distribute. One difference is that distribute sends at most 2 NODEs from its OPEN list to any neighbor. The algorithm for distribute is:

distribute
If (there is enough work to distribute) send NODEs to nearest neighbors
END*distribute*

The other function, *receive_dist*, receives the distributed NODEs and inserts them into the OPEN list. The algorithm is:

receive_dist
Receive NODEs from neighbor
Insert NODEs into OPEN list
END*receive_dist*

Additional variables are needed to help control when to distribute NODEs. SC represents the number of nodes expanded since the last time NODEs were distributed and δ is the minimum

number of nodes a processor must expanded before distributing again. The distributed list with load balancing and NODE distribution algorithm is:

TSP Worker With Load Balancing and distribution Design
Receive cost matrix from host
Perform depth first search on one node to determine initial BEST.cost
Generate starting node in search tree
Distribute children of initial node among all processors
While (not terminated by DONE message)
While (OPEN not EMPTY)
Remove NODE from OPEN queue
Check for RING
If ($SC \geq \delta$) then distribute NODEs to nearest neighbors
Check for distributed NODEs from neighbors
Check for WORK_REQUEST message from another processor
Loop until all cities have been checked
Add a city to end of cities in NODE.vector which have been visited
If new city has not been visited in this partial tour
Calculate the cost ($h(n)$ and f) for new partial tour
If (new NODE.cost < BEST.cost) and (NODE.vector is a tour)
BEST = NODE
Broadcast NEW_BEST to all processors
If (NODE.cost < BEST.cost) and (NODE.vector is not a tour)
Insert new NODE into OPEN
END (Loop until all cities have been checked)
END While (OPEN not EMPTY)
Send WORK_REQUEST to other processors
Terminate the processors
END Loop while (not terminated by DONE message)
Send results to Host
END *TSP Worker With Load Balancing and Distribution Design*

4.6 Summary

This chapter described and provided examples of the data structures used by the algorithms. The high level design of the TSP algorithm was further developed by adding communication requirements and architecture specific details. Algorithms of the functions used by the main algorithms are discussed. Changes to the basic parallel TSP algorithm including delayed A* and distributed list queues with and without load balancing are discussed and the algorithms provided and explained.

Also discussed were two variations to the basic A* algorithm. The first algorithm, IDA*, performed a limited depth first search in conjunction with the A* algorithm. The continuous diffusion algorithm attempts to insure the nodes being expanded are not wasted work by exchanging NODEs between processors.

The next chapter provides and discusses the results of all the algorithms.

V. Results

5.0.1 Introduction The previous two chapters presented the design of the sequential TSP using the A* algorithm. Also presented were parallel algorithms of the TSP using A* including centralized list, distributed list with and without load balancing, IDA*, and continuous diffusion. The CL algorithm was developed first and then tested. While this algorithm is efficient for a small number of processors, the other algorithms were developed to reduce the amount of idle time on the processors or to reduce the number of states explored by the algorithm.

The purpose of this chapter is to discuss the data gathered while executing these algorithms. Section 5.2 discusses the metrics used to gather and evaluate the data during testing of the algorithms. Also provided are the test cases against which the programs were run. Section 5.3 discusses how to read the test results. Only the results of the programs are presented in this chapter. The evaluation and interpretation of the results is presented in Chapter VI.

5.1 Metrics

This section is a further discussion of the metrics presented in Chapter III and provides a detailed explanation of the metrics used in this research.

When gathering data on a program, it is initially difficult deciding how much and of what type of data to collect. Too much data can swamp someone trying to evaluate it and they might miss something of importance. Too little data and something of importance might not be reported. Also, the amount of data data collected can have an adverse impact on the performance of the program. While there was a basic core set of parameters that had to be measured only once, the frequency of other parameters was determined on a trial and error basis. The first few runs of the centralized list program produced too much information. A relatively small program took minutes to run compared to the seconds it took to run after the parameters were tuned.

The specific parameters used to evaluate the TSP A* programs are listed below:

- Total program run time — The total execution time of the program, from initiation to termination.
- Initiation time — Time spent loading the cost data and initializing the variables.
- Search time — The time spent searching for the solution. This includes communication and idle time. It is calculated by

$$Searchtime = Totaltime - Initializingtime$$

- Processor efficiency — The ratio of the time a processor was in the search portion of the program versus the total execution time. The search portion of the algorithm does not include any idle or communication time.
- Average processor efficiency — The average of the processor efficiencies. One or two processors could have a low efficiency, but the overall efficiency of the program could still be high.
- States expanded per processor — The number of states expanded per processor. This metric can indicate idle time or inefficiencies in distributing the work.
- Total states expanded — The total number of states expanded. This is one of the best metrics for comparing search algorithms.

Other parameters that are not used to evaluate the algorithms but were used for troubleshooting purposes were:

- NEW_BEST — A printed message indicating a new global best solution was found and by which processor.
- Queue size — A printed message showing the size of the OPEN list on the Worker's or the centralized OPEN list on the Controller. Also printed the cost of the next NODE to be

expanded. This parameter was very helpful in determining the effectiveness of the prune function.

These parameters also provided a *feel* of how the programs were running. Several times problems were detected just by the program not acting as it had in the past.

The IDA* algorithm had the following unique parameters:

- IDA_NEW_BEST — A printed message indicating a new global best solution was found during the IDA* portion of the algorithm and by which processor.
- IDA_expanded — Number of states expanded during the IDA* portion of the algorithm.

Parameters unique to the distributed list programs are:

- Distributed — The number of times a processor distributed work to its nearest neighbors
- Asked_for_work — The number of times a processor became idle and requested work from another processor.
- Share — The variable used to adjust when a processor had enough work to share with another processor. This variable was used in both the "share_work" and the "distribute" functions.
- Lambda — The variable used to determine when a processor should distribute NODEs to its nearest neighbors.

5.2 Testing

The main measure of an algorithm is that it produces the correct results. As the example in Chapter I showed, even relatively small problems can have a prohibitively large number of combinations which must be checked. The method used to validate the program results was to run the program and use the results as the "best" solution. With this solution as the bound, problems

with 4, 10, and 22 cities were then solved by hand. For example, Figure 5.1 shows the configuration for the 4 city input. Figure 5.2 shows the search graph generated by hand to solve this problem.

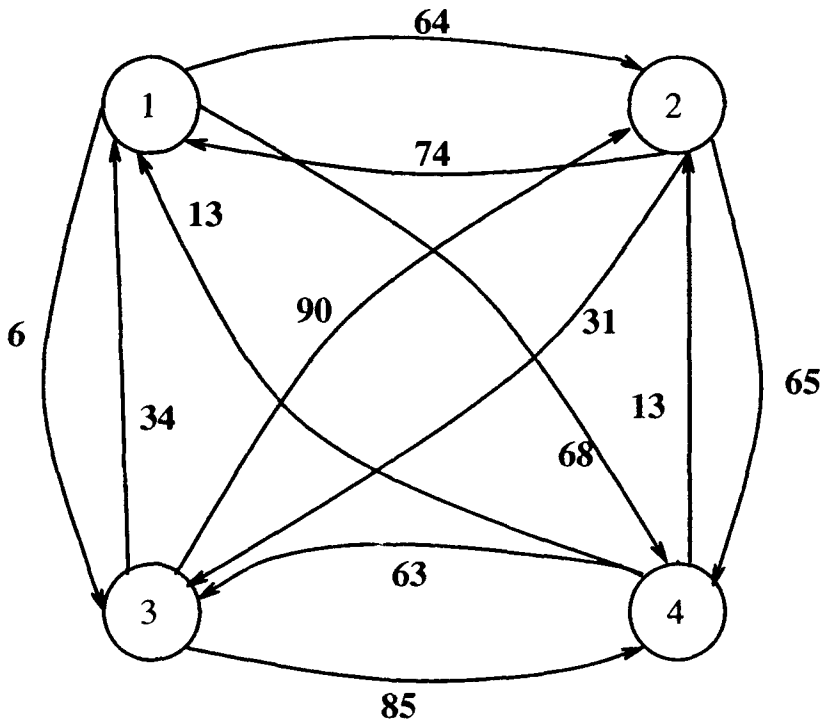


Figure 5.1. TSP for 4 Cities using File n4a

While the 4 and 10 city problems were relatively easy to solve, the 22 city problem was selected because it was the about the largest problem solvable by hand in a reasonable amount of time. Even having the best bound possible, i. e. , the solution, this problem took over 4 hours to solve.

In each case, the solution generated by the program and the solution generated by hand had the same best cost. Since it is impossible to check problems of any size by hand and the same test cases were run using different algorithms and parameters, the assumption is made that the correct answer is produced if it matches the answer given by different algorithms running the same test case.

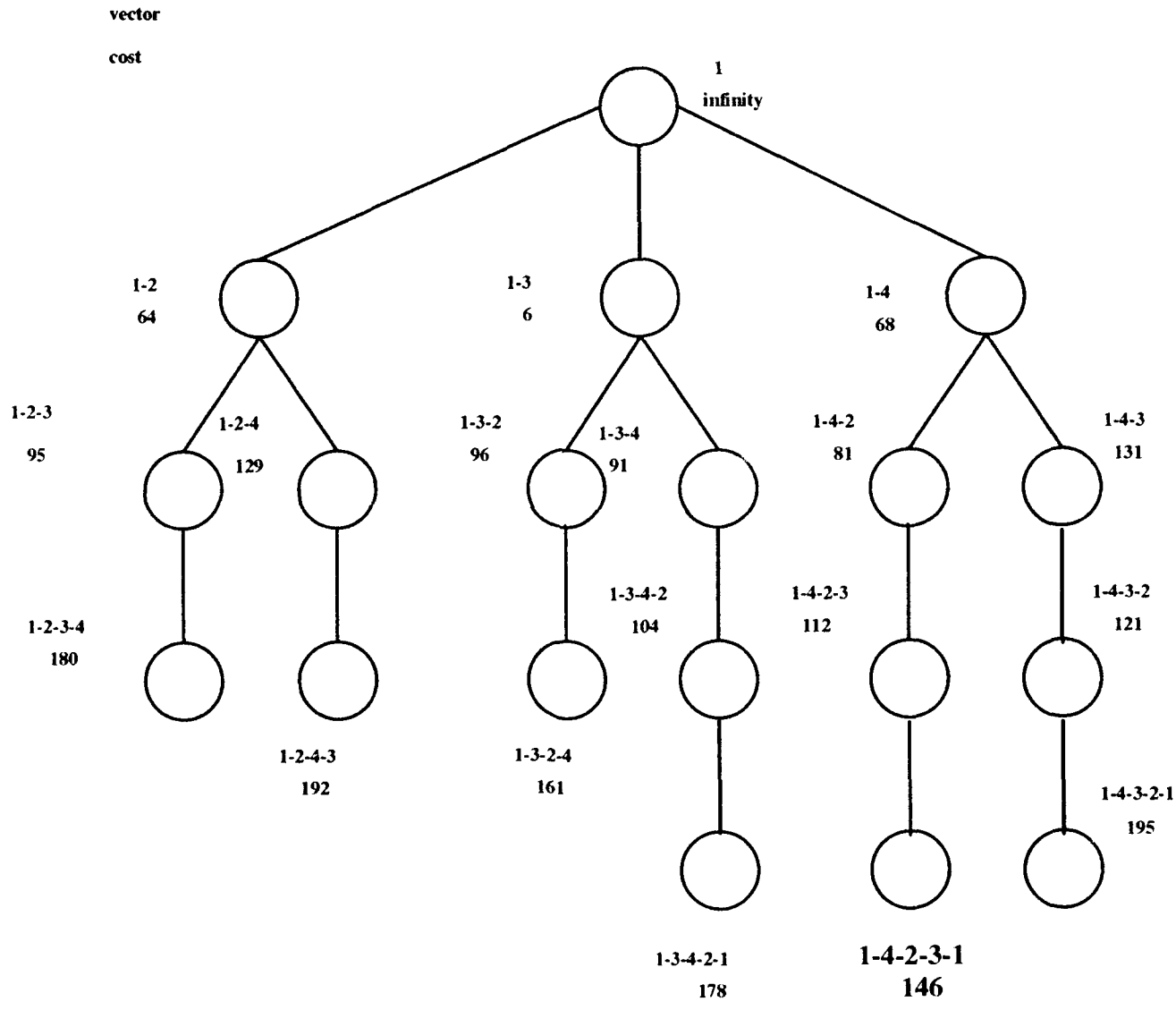


Figure 5.2. Search Graph for 4 Cities using File n4a

The distributed list with load balancing and the continuous diffusion algorithms required special testing because they have parameters which must be *tuned* for each application. Therefore there are several different runs of the same program with the parameters changed. A listing of the test case inputs and solutions for all the problems are provided in Appendix C

Each test case was run using all the algorithms described in Chapter IV on 2, 4, 8, 16, and 32 processors. Since the Air Force Institute of Technology (AFIT) has only an 8 processor hypercube, another hypercube with 64 processors was located at Oak Ridge National Laboratory, Oakridge, TN. While this hypercube had enough processors for the tests, other problems arose. First, the network used to connect to the Oakridge hypercube, the Defense Data Network (DDN), had routing problems. The software routines to route the telephone connections had been recently modified and no reliable path between AFIT and Oakridge could be found. Response times of up to 10 minutes for each keystroke were noted. This problem was finally partially solved by logging into a computer at Phillips Lab in *Albuquerque, NM* then logging into the Oakridge hypercube. This produced a response time of about 2 seconds which was acceptable, but the computer at Phillips Lab was down for maintenance frequently.

Another problem with the Oakridge hypercube was the operating system of the iPSC/2 is not very robust and can "crash" quite easily. When working with the AFIT iPSC/2, the status of the computer can be monitored by watching the status lights on the front of the computer. Also, the system administrator, Richard Norris, was *very* helpful in determining the cause of the crash and ways to fix it. Since the iPSC/2 at Oakridge was remote, the status lights could not be monitored.

5.3 Test Results

This section provides a listing of the data results and what each table measured. The tables along with graphs of data from the tables are provided in Appendix B.

- Appendix C — List the test cases and the solution to each one. The solution is the order in which the cities are visited and the associated cost.

- TABLE B1 — Centralized list program test results, including execution time, states expanded, and average processor efficiency. The entry for one processor is the data for the sequential algorithm.

- TABLE B2 — Distributed list without load balancing program test results, including execution time, states expanded, processor efficiency, and average processor efficiency.

- TABLE B3 — Distributed list with load balancing program test results, including execution time, states expanded, and average processor efficiency. Also included are the number of times the processor asked for work.

- TABLEs B4 and B5 — Distributed list with load balancing and distributing program test results, including execution time, states expanded, processor efficiency, and average processor efficiency. Also included are the number of times the processor asked for work and distributed NODEs to its nearest neighbors.

- TABLEs B6 and B7 — Distributed list with load balancing program test results, including execution time, states expanded, processor efficiency, and average processor efficiency. Also included are the number of times the processor asked for work and distributed NODEs to its nearest neighbors. This table differs from TABLE B3 in that it shows the optimal range for the share variable.

- TABLEs B8 and B9 — Distributed list with load balancing and distributing program test results, including execution time, states expanded, processor efficiency, and average processor efficiency. Also included are the number of times the processor asked for work and distributed NODEs to its nearest neighbors. This table differs from TABLEs B4 and B5 in that it shows the optimal range for the distribute variable.

- TABLE B10 — IDA* program test results, including execution time, states expanded, and average processor efficiency. Also included are the number of states expanded during the IDA* portion of the program.
- TABLE B11 — Level program test results, including execution time, states expanded, and average processor efficiency. Also included are the number of states expanded during the level portion of the program.

5.4 Summary

This chapter provided the results from all the algorithms developed and tested. The metrics used to evaluate and compare the algorithms are also discussed. Due to limitations in time and the adverse impact on program execution, data on all possible metrics were not collected.

The sequential version of the TSP using A* was shown to return the correct results on known test cases and problems of small enough size to be evaluated by hand. Therefore, the results of the sequential program are assumed correct. No parallel version of the algorithms returned a cost different than the sequential program.

The next chapter evaluates the results from the different algorithms. Conclusion about the efficiency of the algorithms are drawn and recommendations for further work is presented.

VI. Conclusions and Recommendations for Further Work

6.1 Introduction

Chapter I describes the nature of NP-complete problems and provides an example of the exponential nature of the time and polynomial nature of the space requirements of these problems. Also described are some of the physical constraints and limits on the capabilities of sequential computers. These limitations coupled with the increasing power and decreasing cost per million instruction per second (MIPS) have led to the use of parallel computers for large, complex problems.

Chapter II provides the background for this research investigation. A more detailed definition of NP-complete is given along with the relationship between NP-complete, P-time, NP-time, P-space, and NP-space. Parallel computers in general and the hypercube specifically are discussed. Some of the problems related to parallel programming of search algorithms are listed and briefly explained. Finally, different search techniques are described. Which technique or combination of techniques to use is problem specific and Table 2.2 provides a general listing of the strengths and weakness of each algorithm.

More specific background information relating to metrics used to evaluate parallel algorithms and specifically parallel search techniques is discussed in Chapter III. Metrics such as processor idle time, program run time, and number of states expanded were chosen for their ability to measure and show the relative efficiency of the programs. The assignment problem and how it relates to the implementation of the traveling salesman problem (TSP) is discussed and an example of its use given. Finally, the high level design of the parallel A* TSP algorithm using a centralized list (CL) is discussed. Pseudo code for the algorithm is provided along with a discussion of each part of the algorithm.

Chapter IV discusses the low level design of the parallel A* TSP algorithms. High communication overhead and poor scalability due to the master processor becoming a communication bottleneck in the CL version of the TSP algorithm led to the development of distributed list (DL)

versions of the parallel TSP algorithm. When to share or distribute work are the main areas of study for these algorithms. Also, an IDA* version of the TSP algorithm is developed. For each of these algorithms, the pseudo code is given and a detailed discussion of the algorithm is provided.

Chapter V presents the results from each algorithm, including which metrics are used for evaluating the use of the synthesized program and the testing process. A listing and explanation of how the data is displayed is also provided. Interpretation of the results is left for this chapter.

This chapter has two main objectives. First, the data from each of the algorithms is interpreted and compared to the other algorithms. Specifically, the three main metrics of program run time, states expanded, and processor idle time, are compared and explanations for the differences are provided. Graphs of applicable data are provided to substantiate the explanations for the differences among the algorithms.

During any time limited research effort, it is never possible to explore all avenues of interest or problem areas. Therefore, the other objective of this chapter is to provide recommendations for further study.

6.2 Interpretation of the Results

In this section, the results from the four main algorithms investigated by this research, are discussed. These algorithms are:

- Centralized list A* TSP
- Distributed list with no load balancing A* TSP
- Distributed list with load balancing A* TSP
- Distributed list with load balancing and work distribution A* TSP

Abderlrahman and Mudge [Abderlrahman and Mudge, 1988: 1497], Quinn [Quinn, 1990: 385-387], and others have shown, the master processor in a worker/manager functional decomposition

using a centralized list can become a communications bottleneck. They also state that after a certain number of processors, increasing the number of processors actually increases the execution time due to this bottleneck. For the Intel iPSC/2 hypercube, the "magic" number of processors is about 16 and this is validated by this research. Therefore, the algorithms are compared twice, once using 16 or less processors, called small scale computers, and once using more than 16 processors, called large scale computers.

6.2.1 Preliminary Depth First Search (DFS) The high level design for the Control CL version of the parallel TSP algorithm performs an initial DFS of one node. This is done by traversing all cities in numerical order. For example, a 10 city problem initial solution is 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 1. This arbitrary tour provides an initial best solution used to bound the search process. Even for the 100 city problem, this DFS took less than 1 millisecond.

Initially, using the DFS appeared to have no effect on the time of the search or the number of states expanded. However, it was found that on some problems the run time and number of states expanded were greatly reduced. For example, the 65 city problem in file "n65a" expands 1697 states and takes 4153 seconds to run using the DFS on 4 processors. Without the DFS, it expands 3509 states in 8653 seconds. Obviously, the possibility of wasting less than 1 millisecond is worth the large gains the DFS might provide. Therefore, DFS is used with all algorithms developed for this research.

Another variation not investigated by this research is to have each processor perform a different DFS and use the best solution as the initial bound. Also, each processor could perform several depth first searches before continuing on with program execution. The tradeoff between time spent in DFS versus the time saved by finding a "good" initial bound need to be investigated.

6.2.2 Evaluation of the Algorithms In evaluating any algorithm, two main metrics are used. First, an algorithm must be effective in that it provides the correct answer. Secondly, an algorithm

is evaluated as to its efficiency in both time and memory requirements. This research does not investigate methods to improve or compare memory usage.

The three main efficiency metrics used in this research to evaluate an algorithm's execution time efficiency are total execution time, number of states expanded, and processor idle time. Of these, the most important is execution time. For non-research problems, the bottom line is providing a correct solution in the shortest time possible, or at least in an acceptable time. So the overall goal is to find algorithms which perform tasks quicker. While there is normally a direct correlation between the number of states expanded, processor idle time, and the execution time of a search algorithm, states expanded and idle time are still useful metrics. The number of states expanded can be used to compare different algorithms or the same algorithm run on different computers. Processor idle time is helpful in showing where a parallel algorithm is inefficient and possibly how to improve it. These metrics are the basis for comparing the algorithms.

For all algorithms discussed, four different problem sizes are run. These four problems are representative of the different size problems likely to be encountered and contain 22, 55, 65, and 100 cities. Descriptions of the problems along with the cost matrices and problem solutions are in Appendix C. Results such as execution time, number of states expanded and processor efficiency are provided in Appendix B. Some of the graphs of data are also provided in the appropriate sections for ease in understanding the discussion.

6.2.3 Small Scale Parallel Computers This section compares the different algorithms using 16 or less processors on an Intel iPSC/2 hypercube. Each of the four algorithms are discussed and compared to the others.

6.2.3.1 Centralized List Algorithm (CL) The first thing to notice is that for all cases, the CL algorithm outperformed all others in both execution time and in the number of states expanded. This is the same results obtained by Abderlrahman and Mudge [Abderlrahman and

Mudge, 1988: 1497], Quinn [Quinn, 1990: 385-387] and many others. This is because in small scale computers, the master processor does not become a communication bottleneck. This allows for relatively efficient load balancing of the processors and reduces the processor idle time. Also, since the OPEN list is kept on a single processor, the order in which the states are expanded closely resembles the sequential algorithm. The graphs in Appendix B show the number of states expanded per processor is almost a horizontal line for the CL algorithm. As discussed in Chapter III, this means relatively few states not expanded by the sequential algorithm are expanded resulting in little wasted work.

6.2.3.2 Distributed List With No Load Balancing (DL_NLB) The distributed list with no load balancing (DL_NLB) algorithm performed the worst of all the algorithms in relation to execution time and number of states expanded. Again this conforms with what others have found [Abderlrahman and Mudge, 1988: 1497] [Quinn, 1990: 385-387] [Felten,1988:1501] [Hayes and Mudge, 1989: 1839] [Cvetanovic and Nofsinger, 1990: 86-89]. If it was known along which path a solution could be found, only that path would be explored. But since this is not a greedy algorithm, all paths must be explored implicitly or explicitly. Also, it can not be determined which path will generate the most work, so there is no way to evenly allocate the work load at the beginning of the algorithm. Because the DL_NLB algorithm only divides the work once at the beginning of the algorithm and never balances the load again, once a processor finishes its assigned work, it sits idle until all other processors finish. Therefore, the DL_NLB execution time is the time of the longest path to a solution or bound of the search.

In a DL_NLB algorithm, the only way speedup can occur is by dividing the work among enough processors so a "good" boundary solution is quickly found. This allows the algorithm to prune or eliminate numerous search paths. As the figures in Appendix B show, there are initial drops in the execution times, but then the times are almost constant. Adding more processors does not noticeably decrease the execution time because the run time limit is the time of the longest

solution or bound. Adding more processors to solve the problem reaches the longest path quicker, but do not help decrease the time to explore the path.

6.2.3.3 Distributed List With Load Balancing (DL_LB) As discussed in Chapter II, one way to reduce the algorithm execution time when using a distributed list algorithm is to allow processors to request work from another processor when they become idle. One problem with this approach is the overhead associated with determining if a processor has enough work to share and the communication to pass work between processors. These problems are discussed in greater detail in section 6.2.5.

Figures B1, B2, B3, and B4 show the problem size has a real impact on the execution time of the DL_LB. For example, the 22 city problem actually increased in execution time going from 2 to 4 processors then decreased again from 4 to 8 processors. The 22 city problem at times ran slower than the DL_NLB algorithm and never dramatically decreased the execution time. The 55 city problem also ran approximately the same amount of time as the DL_NLB algorithm. But looking at the 65 and 100 city problems, you notice dramatic decreases in execution times. For example in the 100 city problem using 4 processors, the times decreased from 33504 to 20117 seconds. This is a decrease of 40 %!

This discrepancy between small and large sized problems is explained by the communication and task granularities of the different problems. As stated in Chapter II, granularity is a measure of relative size or frequency of an event or computation. In the small problems, the amount of overhead associated with sharing work far outweighs the gain provided by not having idle processors. The problems are so small that it is more efficient to solve them using the CL algorithm or even a single processor.

The two large problems are more representative of the size of problems that would be solved using a parallel computer. For these problems, the DL_LB algorithm performs better than the DL_NLB algorithm but not as good as the CL algorithm. The idle time associated with the

DL_NLB algorithm is greatly reduced, but the overhead associated with load balancing still makes this algorithm less efficient than the CL algorithm. The number of states expanded by this algorithm is also much higher than for the CL algorithm. This is due to each processor having its own local OPEN list. This means only a local, not global, best state is selected for expansion resulting in wasted work.

6.2.3.4 Distributed List with Load Balancing and Distribution (DL_DIST) The DL_DIST algorithm is identical to the DL_LB algorithm with the addition of a function to distribute work from its OPEN list to its nearest neighbors. This is an attempt to emulate the global OPEN list of the CL algorithm and reduce the number of states expanded. For the small problems, the DL_DIST performance is better than the DL_LB algorithm. For the larger problems, the performance is mixed. In the 65 city problem, both the execution times and number of states expanded are better for the DL_DIST algorithm than the DL_LB algorithm. However, in the 100 city problem, the DL_LB algorithm executes in less time until about 7 processors are used. Using between 7 and 16 processors the DL_DIST algorithm performs better. In both cases, the differences are not very great. Again, the DL_LB and DL_DIST algorithms are discussed in section 6.2.5.

6.2.3.5 Small Scale Parallel Computer Summary For the small scale computer, the CL algorithm performed better than the other algorithms. However, when using 8 processors or more, the master processor begins to become a bottleneck and the execution time becomes almost constant. Increasing the number of processors does not decrease the execution time. The DL_NLB algorithm performed the worst of all the algorithms due to load imbalances resulting in processor idle time. Both the DL_LB and DL_DIST algorithms did not perform as well as the CL algorithm, but at 16 processors the execution time curves are beginning to approach the CL execution time curve. When using either the DL_LB or DL_DIST algorithms, adding processor greatly decreased the execution times. The DL_LB and DL_DIST algorithms are discussed in more detail in section 6.2.5.

6.2.4 Large Scale Parallel Computers This section discusses the behavior and performance of the algorithms using 16 or more processors to solve the problems. Because of hardware problems with the Oak Ridge Nation Laboratory's iPSC/2, not all algorithms were able to be run using 32 processors for all problem sizes. However, enough data was collected to show the overall trends of each algorithm.

6.2.4.1 Centralized List As discussed earlier, the CL algorithm's main deficiency is that the master processor becomes a communications bottleneck forcing slave processors to remain idle waiting for new states to expand. This trend is painfully obvious when using more than 16 processors. The execution time curves for all problem sizes had already begun to flatten out when using between 8 to 16 processors. Increasing the number of processors provided no noticeable decrease in the execution times for the problems, and in the case of 100 cities the execution time began to increase.

One important factor to note when comparing the different algorithms is the processor efficiency versus the number of states expanded. The CL algorithm always expanded the fewest number of states, but its efficiency really drops off as more processors are used to solve the problem. The distributed list algorithms expand many more states than the CL algorithm and their processor efficiencies are relatively high. For example, in the 100 city problem, the CL algorithm's efficiency decreases from 0.938 to 0.422 when using 4 and 16 processors respectively. However, the DL_DIST algorithm's efficiency only drops from 0.997 to 0.988. Since the CL algorithm's execution time is always less than the DL_DIST algorithm's, this implies one of the most important factors for reducing search algorithm execution time is not processor efficiency, but the heuristic used to determine the order in which the search graph is explored!

6.2.4.2 Distributed List With No Load Balancing For the same reasons discussed in the small scale computer section, the DL_NLB algorithm execution time curve is almost a constant when using more than 16 processors. Like the CL algorithm, increasing the number of processors

does not decrease the execution time. Also like the CL algorithm, the 100 city problem execution time increased when going from 16 to 32 processors. The minor changes required to add load balancing to this algorithm are definitely worth the cost.

6.2.4.3 Distributed List With Load Balancing and Distribution This section discusses both the DL_LB and DL_DIST algorithms. A comparison of these algorithms is provided in section 6.2.5. While only the 100 city and 65 city DL_LB problems were able to be run using more than 16 processors, the trends are obvious. The execution time curves had already begun to flatten out when using less than 16 processors. When using more than 16 processors this trend continues and the curves flatten even more. Processor idle time has increased due to the overhead and communication associated with load balancing. As the number of processors increases, the more likely it is for a processor to request work from an idle processor or one that does not have enough work to share. However, since the CL algorithm execution times have become almost constant, the DL_LB and DL_DIST algorithms' execution times are approaching the CL times. For the 100 city problem, the difference between the DL_DIST and CL algorithm is only 264 seconds or 9%. This much variance in execution time was noted when running the exact same problem repeatedly.

The number of states expanded by the DL_LB and DL_DIST algorithms increases dramatically with the number of processors used. This is due to the OPEN list being local to each processor and not global. As discussed in Chapter III, this results in expanding states which are not expanded when using a global OPEN list. This tradeoff of wasted work versus reducing the execution time is acceptable, especially since the distributed list algorithms are scalable to large numbers of processor.

6.2.4.4 Algorithm Summary For all algorithms and problem sizes, the CL algorithm still has the shortest execution time. However, as the number of processors increases above 16, the DL_LB and DL_DIST algorithms' execution times continue to decrease while the CL execution time is almost constant. At approximately 32 processors the execution times of the CL and both DL algorithms meet. After this point, the DL algorithms should be more efficient, but no computer was

available to validate the continuation of the algorithm execution time curves. Using the number of states expanded as the metric, the CL algorithm again is the best.

6.2.5 Comparison of DL_LB and DL_DIST Algorithms Many articles discussed distributed list load balancing and distributing nodes from the OPEN list in an attempt to emulate a global list. However, the only guidance provided by any article was to state that determining when and how to balance the load was very difficult and each algorithm or problem was unique [Abderlrahman and Mudge, 1988: 1495] [Quinn, 1990: 385-387] [Felten,1988:1496] [Hayes and Mudge, 1989: 1836] [Cvetanovic and Nofsinger, 1988: 86-89] This section discusses the differences between the DL_LB and DL_DIST algorithms and provides some general guidelines and observations about when and how to balance the work load when using a distributed list algorithm. Most of the data gathered was using 16 or less processors because of the hardware problems with the Oak Ridge National Laboratory's iPSC/2 hypercube.

Since there are two variations of the algorithm investigated by this research, there are 4 combinations of the variations. The combinations are:

- Distributed list with load balancing (DL_LB) — The term *share* is used to describe passing work from one processor to another for the purpose of balancing the work loads. Sharing work is only initiated when a processor is idle.
- Distributed list with distribution but no load balancing (DL_LB) — The term *distribute* is used to describe passing work from one processor to another for the purpose of emulating a global OPEN list. Distributing work can be done any time during program execution.
- Distributed list with load balancing and distribution (DL_DIST) — This algorithm uses both load balancing and distribution. This algorithm also encompasses the fourth variation of using distribution and load balancing.

The distributed list with distribution and no load balancing is very similar to the distributed list with no load balancing. The differences are examined when the DL_LB and DL_DIST algorithms are discussed.

6.2.5.1 Load Balancing As stated in Chapter IV, there are two aspects to sharing work. First, a processor must be idle and second, another processor must have enough work to share. The variable *share* is used to determine the number of nodes required on a processor's OPEN list before it can share work with another processor. As stated in Chapter II, there are tradeoffs to consider when determining when and how to share work. Sharing work too often and the communication overhead negates any advantage from sharing; don't share enough and processors remain idle!

Looking at the data in Appendix B Tables B6 and B7, the first thing to observe is the two smaller problems have the same share variable and the two larger problems have the same share variable. This indicates the problem size has an effect on how often to share. The larger problems require more time to determine if the next city added to the partial tour is already in the path, are the cities now a tour, and calculate the estimated cost to complete the tour. This means there is a relatively large amount of computations required for the larger problems. Therefore, once work is shared, it takes a longer period of time before the large problems would request work again. This allows the share variable to be set smaller and optimize the computation to load balancing overhead ratio. When the share variable is too small, a processor sends work to a requesting processor and then quickly finishes its own remaining work. Now the processor is idle and must request work. This cycle continues with relatively few states being expanded and a relatively large time spent in load balancing. This is similar to *thrashing* where a processor is constantly requesting data from secondary memory and little computation is performed. *Share thrashing* occurs when load balancing overhead dominates the computations performed to solve the problem.

Another factor effecting the share variable is the number of children produced by a problem. In this research, the smaller problems produced fewer children. For example, the 20 city problem produces at most 20 children while the 100 city problem produces at most 100 children. This means once a large problem receives work, it is more likely to produce children and need not request additional work immediately. Problems which generate small amounts of additional work need to share less frequently, but share larger blocks of work. If a problem generates few children, share thrashing could occur if the share variable is set too low.

Especially for the large problems, there is a large decrease in the execution times between the optimal share value and the next lower value. For example, the 100 city problem has an execution time of 27,838 seconds for a share value of 3 and 9,210 seconds for a share value of 4. When the share value was set at 2, both the 100 and 65 city problems were terminated after running 24 hours and did not appear close to finishing. The additional overhead of load balancing should not account for such a large increase in execution times, especially since the problems require relatively large computation time and generate a large number of children. What was determined was that while the computation time required to expand a state remained approximately constant, the number of children generated did not. This is due to two reasons. First, as Figure 3-10 shows, the number of children generated decreases at each level of the search tree due to reduced combinations of solutions available. The second reason is that as the algorithms progress, the cost used to bound the solutions becomes closer to the optimal solution. This increasingly eliminates children which are generated from being placed on the OPEN list because their estimated cost to completion is already higher than the current best solution. After the bounding cost gets relatively close to the optimal cost, the large problem behaves like a small problem as far as child generation is concerned. This is why the optimal share variables are so close for both the large and small problems and the execution times vary so drastically with a small change in the share value.

This situation where the large problems act like small problems lends itself to a graduating scale approach to selecting the share variable. At the beginning of the program, the share variable can be relatively low with the value increasing as the bounding cost approaches the optimal cost. When to increase the share value could be determined by the percent of possible children actually generated for expansion. The higher the percent, the lower the share value.

A third factor effecting the choice of a share variable is the processor computational speed. As the computational speed increases, work should be shared less often. This is because the processors can quickly expand the states. If the share variable is too low, share thrashing occurs. An example of this is comparing the iPSC/2 hypercube to the i860 hypercube using identical problem sizes, algorithms, and share values. As explained in section 6.4, the computational speed is approximately 14 times faster for the i860. Using 8 processors, a share value of 7 and the 55 city problem, the iPSC/2 requested work 243 times with a run time of 1103 seconds while the i860 requested work 687 times with a run time of 172 seconds. However, when the i860 share value was increased to 10, the run time decreased to 141 seconds and work requests dropped to 350. While this is not conclusive proof, the trend held for all algorithms and share values tested.

6.2.5.2 Distribution The idea of distributing the workload to achieve greater efficiency is discussed by Felten [Feldman, 1989: 15-0-1504] Cvetanovic and Nofsinger [Cvetanovic and Nofsinger, 1990: 82-90] and many others. Again, the only guidance provided was that each problem is unique and that the optimal distribute variable must be found by trial and error.

In this research study, the distribute variable determines the frequency of a processor sending work to its nearest neighbors. A counter is incremented after every expansion of a state. When the counter equals the distribute variable, work is distributed. Another condition required for distribution is that the number of nodes on the OPEN list be equal to or greater than the share value of the variable. This ensures a processor does not distribute work and then have to request work.

The DL_DIST algorithm has all the advantages and disadvantages of the DL_LB algorithm with the addition of the distributing of work. In these algorithms the DL_LB algorithm had the most impact on reducing execution time. The DL_DIST algorithm only added a relatively small amount to the reduction in run times. For example, the execution time for the 65 city problem using 16 processors and the DL_NLB algorithm decreased from 13762 seconds to 8901 seconds when the DL_LB algorithm was used. This is a decrease of 35%! However, using distributed list with no load balancing and distribution, the run time only decreased to 12984 seconds, a decrease of about 5%. This shows the predominate factor in reducing run time is balancing the work load to keep all the processors busy.

One goal of distributing work is to more closely emulate the global OPEN list of the CL algorithm. The data in Figures B5, B6, B7, and B8 show the DL_DIST algorithm consistently expands fewer states than the other two distributed list algorithms. Also, the data in Tables B8 and B9 shows that the number of states expanded is inversely proportional to the frequency of work being distributed. This shows that by distributing work more frequently, the closer the algorithm emulates the global OPEN list. Unfortunately, the added distributions also incur additional overhead to process and send the work to other processors. So while the goal of more closely emulating the global OPEN list was met, the additional overhead requires tradeoffs between the benefits of distributing work and the overhead incurred.

Because the load balancing is the dominante factor in reducing program execution time, few observations can be made about the distribute variable. First, if the distribute variable is too low, a problem similar to share thrashing occurred. Little computation was getting done because all the time was spent in distributing work. The distribute thrashing problem is related to the amount of computation required to expand the states. Like the share variable, the more computation required to expand the states, the more frequently work can be distributed.

One problem not mentioned in the literature is not allowing work to be distributed if the cost of the nodes being distributed is constantly the same. For example, using 8 processors and the 100 city problem, the DL_DIST algorithm reaches a point where there are approximately 17,000 nodes on all OPEN lists. Of these 17,000 nodes, approximately 90% have a cost of 134. Therefore, it makes no sense to distribute nodes when all the OPEN lists have the same cost. This problem is reduced by having a counter keep track of the number of times work was distributed with the same cost. If the counter reaches a predetermined number, no work is distributed until the cost of the node on the front of the OPEN list changes. For this research, the number was set at approximately 50. This number was selected to allow nodes of the same cost to be spread among all the processors, but keep nodes from being unnecessarily distributed. While the effect on the algorithm was not great, it did reduce the 100 city problem using 8 processors from 13867 seconds run time with 5862 states expanded to 13011 seconds run time and 5844 states expanded. This addition to the DL_DIST algorithm allows problems which generate large numbers of children with the same cost to efficiently distribute work.

Having a large number of nodes with the same cost diminishes the expected reduction in the program run time.

For example, compare the 65 city problem which has few nodes with the same cost and the 100 city problem which has many nodes with the same cost. The most same cost nodes observed when using 8 processors in the 65 city problem was approximately 1200 nodes with a cost of 137 while the 100 city problem 17,000 nodes with a cost of 134. In the 65 city problem, the difference in execution times between the DL_DIST and DL_LB algorithms continues to increase as more processors are added. In contrast the when using the 100 city problem, the DL_LB algorithm never clearly outperforms the DL_DIST algorithm. When using large numbers of processors, the DL_DIST and DL_LB algorithms have almost identical execution times for large nimbers of same cost nodes. This is because by using more processors, the program quickly finds near optimal

solutions and prunes most branches of the search tree. Since there are numerous nodes with the same cost which are within 1% of the optimal solution, this means the nodes left to expand are not distributed. Therefore, the function which keeps work from being distributed when the nodes have the same cost forces the DL_DIST algorithm to emulate the DL_LB algorithm as the number of nodes with the same cost increases.

6.3 IDA Versus Centralized List*

One goal of this research is to compare the IDA* algorithm with the DL_DIST algorithm. For all problem sizes, this implementation of the IDA* algorithm outperformed the DL_DIST algorithm in both execution time and states expanded. This is because the IDA* algorithm is very similar to the CL algorithm and states are expanded in approximately the same order as the CL algorithm. As discussed previously, this is one of the main factors in reducing algorithm execution time and number of states expanded. Since the IDA* algorithm more closely resembled the CL algorithm, it is compared to it instead of the DL_DIST algorithm.

Because of the IDA* algorithm implementation, only the number of states expanded can be compared to other algorithms. The IDA* algorithm consistently expanded more states than the CL algorithm. This is because the CL algorithm uses the assignment problem as the function to estimate the remaining cost to completion for that state. As discussed in Chapter III, the assignment problem can provide a solution to the search problem. This is a form of depth first search that the IDA* implementation does not exploit. To balance the algorithms for comparison purpose, the CL algorithm is required to expand all levels of the search graph. This new CL algorithm is called "level".

Comparing the level and IDA* algorithms produces mixed results. While neither algorithm always expands the fewest number of states, the IDA* does constantly expand less. For example, when using 65 cities, the IDA* expanded as much as 9649 fewer states or 480% less! Whether this

is due to the inherent superiority of the IDA* algorithm or caused by the implementations of the two algorithms could not be determined in this research. Other factors which could effect which algorithm to use and could be investigated include memory requirements for storing the OPEN list and the cost of maintaining an OPEN list versus the cost of repeatedly generating and expanding the states. More study using different implementations of the IDA* and DL_DIST algorithms is required to determine which algorithm is actually better.

6.4 Guidelines for Distributed Memory Computer Implementation of A Algorithms*

This section is a summary of what was learned about implementing A* algorithms on distributed memory computers using data decomposition. This section is composed of three parts: determination of whether to use a centralized or distributed list algorithm, factors effecting the use of a distributed list algorithm, and factors effecting the use of work distribution.

The most important decision is whether to use a centralized list or distributed list to store the states to be expanded. The following are some general guidelines on which list to use:

- Is the problem to be scalable to a large number of processors? If the algorithm is to be scalable to a large number of processors, then use the distributed list. If the algorithm is only going to be run on a small number of processors, then use a centralized list.
- Determine the boundary between centralized and distributed list. Sometimes the number of processors available might be on the borderline between which list is optimal. The determination of which list to use is then determined by trial. However, the number of processors effectively controlled using a centralized list varies and is dependent on the communication/interconnection network and the processor computational speed of the individual computer. The faster the processor, the larger the number of processors efficiently used with a centralized list. For the Intel iPSC/2, the centralized list is efficient up to approximately 16 processors, and possibly more efficient than the distributed list until about 32 processors.

The Intel i860 can efficiently control approximately 40 processors using a centralized list with the computation requiring 40 ms and using the short message protocol [Work, 1991].

The following is a guideline for when using a distributed list:

- Unless there is some special attribute of the problem known during algorithm design to preclude it, load balancing is required to make the algorithm run efficiently.
- The number of states awaiting expansion on a processor before it can share work to load balance is dependent on:
  - Problem size — States awaiting expansion before allowing load balancing are inversely proportional to the amount of computation required to expand one state.
  - Children generated — States awaiting expansion before allowing load balancing are inversely proportional to the number of possible children generated by each state.
  - Graduating scale — Consid - making the states awaiting expansion on a processor before it can share work dependent on at what stage of the program it is in. For example, require a relatively small number of states awaiting expansion at the beginning of the program, increasing the states required as the program progresses. At the end of the program, a relatively large number of states is required for a processor to share work by load balancing.
  - Processor speed — States awaiting expansion before allowing load balancing is directly proportional to the computational speed of the processor.

While the major impact on program execution time is from load balancing, distributing work can also reduce execution time. Factors effecting distribution are:

- Numerous states with same cost - Always have some function to keep from distributing work if the states have the same cost. The minimal overhead of keeping track of the number of

PROCESSOR #	STATES EXPANDED	PROCESSOR #	STATES EXPANDED
1	82	5	37
2	82	6	30
3	58	7	23
4	43		

Table 6.1. States expanded by processor using CL and 100 cities

same cost states distributed is very minimal compared to the cost of removing and inserting the states into the OPEN list and transmitting the states between processors.

- The more computation required to expand a state, the fewer states expanded between work distributions.

While this research did not investigate methods to increase the efficiency of the CL algorithm, a couple of observations were noted. First, slave processor efficiency is inversely related to the frequency of work requests to the master processor. There is a wide variance in the number of states expanded by each processor in solving a problem. For example, the 100 city problem using 8 processors had the distribution in Table 6.1.

Notice the lower the processor number the more states it expanded. While this is dependent on the manner in which idle processors are selected to send work to, this does show a few of the processors are performing most of the work. Processors 1 and 2 expanded 164 out of 355 states or 46%! The individual processor efficiencies also reflect the same trend with processor 1 having the highest efficiency and processor 7 the lowest. This shows processors are waiting for work to be assigned to them. Therefore some method of keeping the frequency of work requests low should be investigated.

Second, the computation speed vs communication speed ratio is a major factor in determining the number of slave processors a master processor can efficiently control. For example, when

using the Intel iPSC/2 hypercube, the master processor controls approximately 16 slave processors before the communications bottleneck does not allow the efficient addition of more processors to the problem. However, the Intel i860 hypercube can control an estimated 40 slave processors [Work, 1991 :] While little data was collected using the Intel i860 hypercube during this research, the data collected does appear to support that a master processor on the i860 hypercube can efficiently control more processors than the iPSC/2. Since both computers have the same interconnection/communication system, the only difference is the computational speed of the processors. Test performed by Richard Norris, the Air Force Institute of Technology iPSC/2 system administrator, show the ratio in computational speed between the i860 and the iPSC/2 is about 14:1. The overall ratio of execution times between the i860 and iPSC/2 is about 8:1. The difference between computation speed and execution times is caused by both computers using the same interconnection/communication network.

6.5 Recommendation for Further Research

In any research effort, there is always work you did not have time to perform and new ideas that evolved but were not explored. The following is a list of topics to further extend this research:

1. Investigate a dynamic algorithm that is a combination centralized list and distributed list. As noted earlier, much of the processor idle time was waiting for the master processor to provide work to the slave processor. One way to alleviate this is to have the slave processor send the low cost children it generates to the master processor and keep a portion of its high cost children generated. This reduces communication costs and provides low priority work for the slave processor while it is waiting for work from the master processor. Another possibility is to have the slave processors which are consistently waiting for work keep a small portion of low cost children for expansion while waiting for the master processor to send work. The

processors which consistently wait for work are easily determined as discussed earlier in this chapter.

2. Since the centralized list is the most efficient for small numbers of processors, investigate ways to optimize the use of this list. Many Air Force requirements in the future will require small parallel computers on board aircraft to perform functions now being performed at the support base or not being performed at all.

3. Investigate other methods of distributing work among processors. One method suggested by Felten is to randomly select processors to distribute work to instead of sending it to the nearest neighbors [Felten, 1988: 504].

4. Determine the number of processors a master processor can efficiently control on the Intel i860 RISC computer.

5. Make the termination sequence for the distributed list more efficient. If a processor is still idle after requesting work from all other processors, it goes into a loop waiting for the RING to come to it to terminate the process. Sometimes,a processor which did not have enough work to share keeps generating children and continues to work long after other processors are idle. After a processor has been idle for a predetermined time, it could force a working processor to share any of its remaining work.

6. Investigate more fully the differences between IDA* and the distributed list algorithms.

6.6 Summary

Because of the combinatoric nature of NP-complete problems, they will continue to be difficult and time consuming to solve. Many factors encourage the use of parallel computers to solve these problems. First, parallel computer costs are decreasing while their performance is increasing. Secondly, these problems have inherent parallelism as demonstrated by the relative ease with

which they are decomposed using data decomposition. Also, it is getting more difficult to increase performance of sequential computers.

This research investigated NP-complete problems on distributed memory architecture computers by implementing a traveling salesman problem using a variation of the A* algorithm. Differences between using a centralized or distributed list to store the states waiting to be expanded were explored. Three distributed list algorithms, DL_NLB, DL_LB, and DL_DIST, were designed and implemented. Advantages and disadvantages of each were discussed and compared to the centralized list algorithm.

The centralized list algorithm is found to perform better than the distributed list algorithms when using a small number of processors. However, this algorithm produces a communication bottleneck and is not scalable to a large number of processors. While the number of processors the master processor can efficiently control is application and computer dependent, some guidelines are given to help determine this value.

Because of the overhead required for load balancing, the distributed list algorithms are not as efficient as the centralized list algorithm for small numbers of processors. However, the distributed list algorithms are scalable to large numbers of processors and become more efficient than the centralized list algorithm. Factors relating to the efficiency for load balancing and distributing the work are discussed. Major factor to load balancing and distributing efficiency include computation required to expand each state, number of children generated by each state, at what stage in the program you are, and the computational speed of the processors.

Appendix A. *Structure Charts*

A.1 Introduction

This appendix shows the relationships between the different functions and subroutines of each algorithm using a structure chart.

A.2 Centralized List Algorithm

The centralized list algorithm consists of three main algorithms, each with subroutines and functions. The three algorithms are the Host, Control, and Worker algorithm,s. The IDA* structure charts are identical to the centralized list charts. The difference in the algorithms is when the NODEs are inserted into the OPEN list. Their structure charts are:

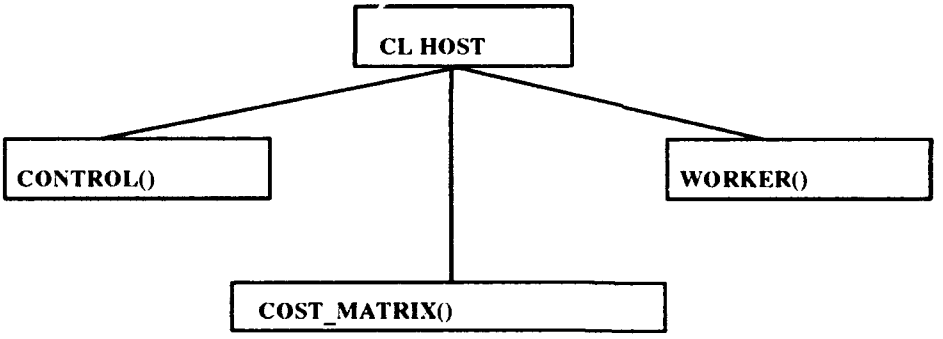


Figure A.1. Centralized List Host Structure Chart

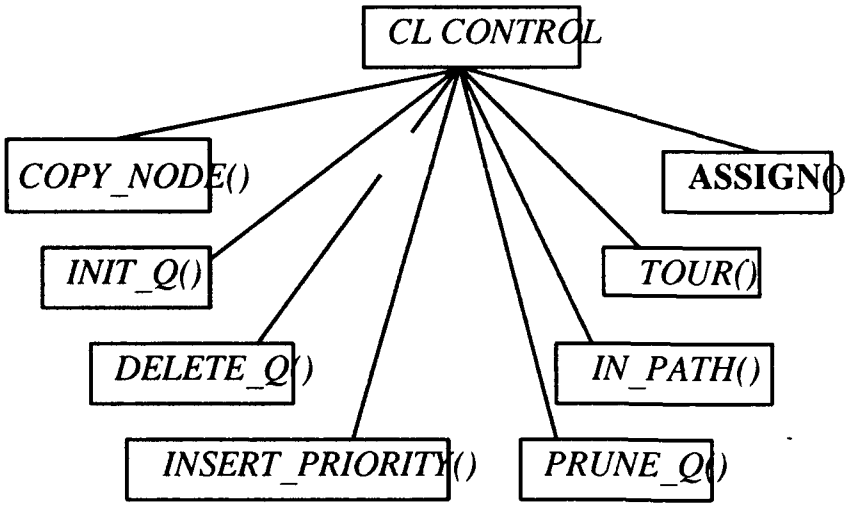


Figure A.2. Control Structure Chart

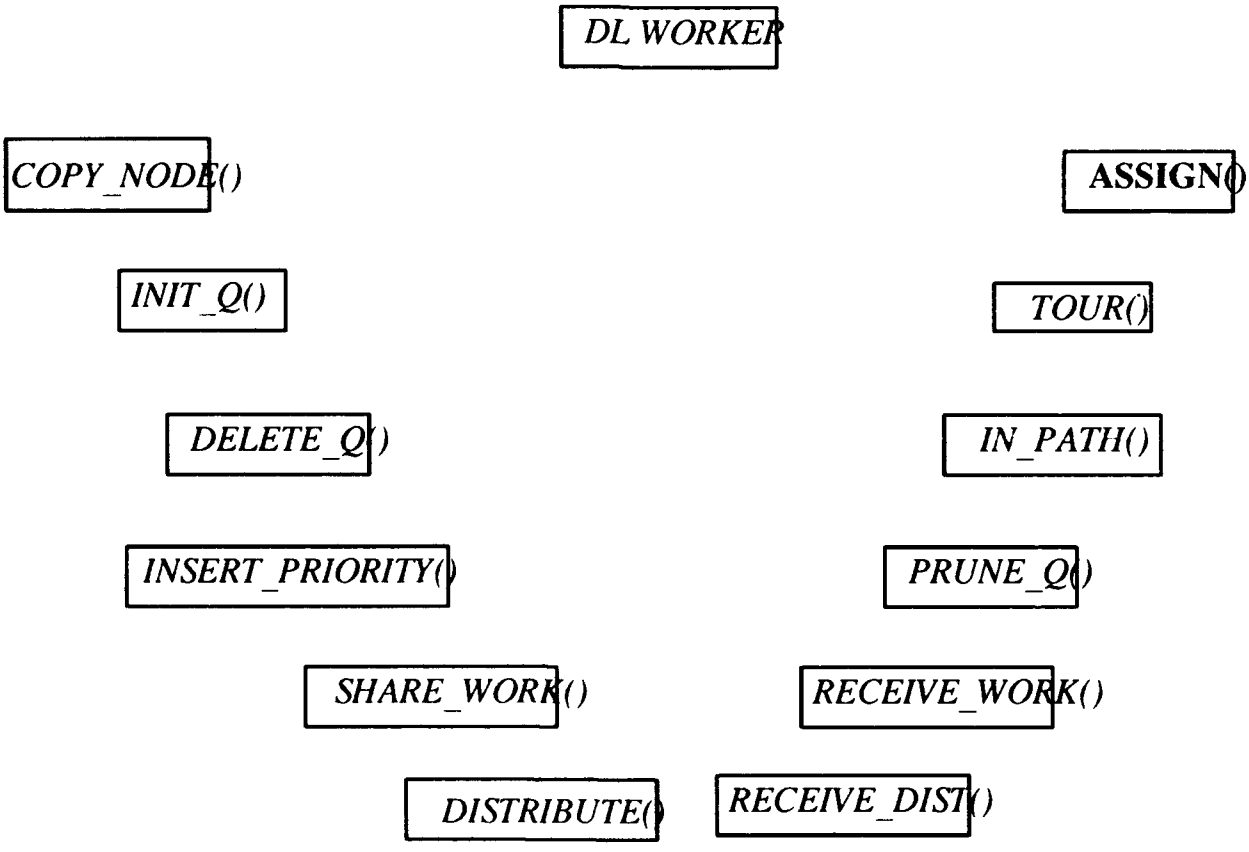


Figure A.3. Centralized List Worker Structure Chart

A.3 Distributed List Algorithms

The distributed list algorithms' structure charts are shown below are each shown below. Since the host algorithm is the same for all variations of the algorithm, it is only shown with the distributed list with load balancing.

A.3.1 Distributed List with Load Balancing The following structure charts are for the distributed list with load balancing algorithm:

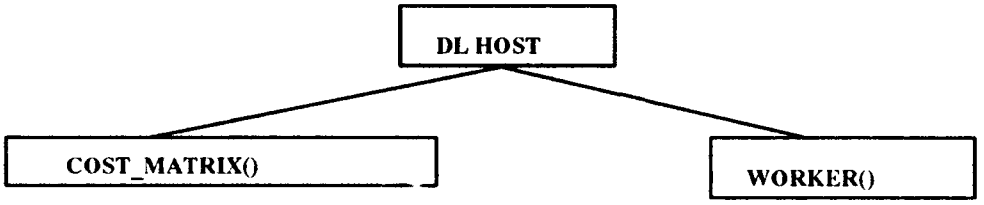


Figure A.4. Distributed List Host Structure Chart

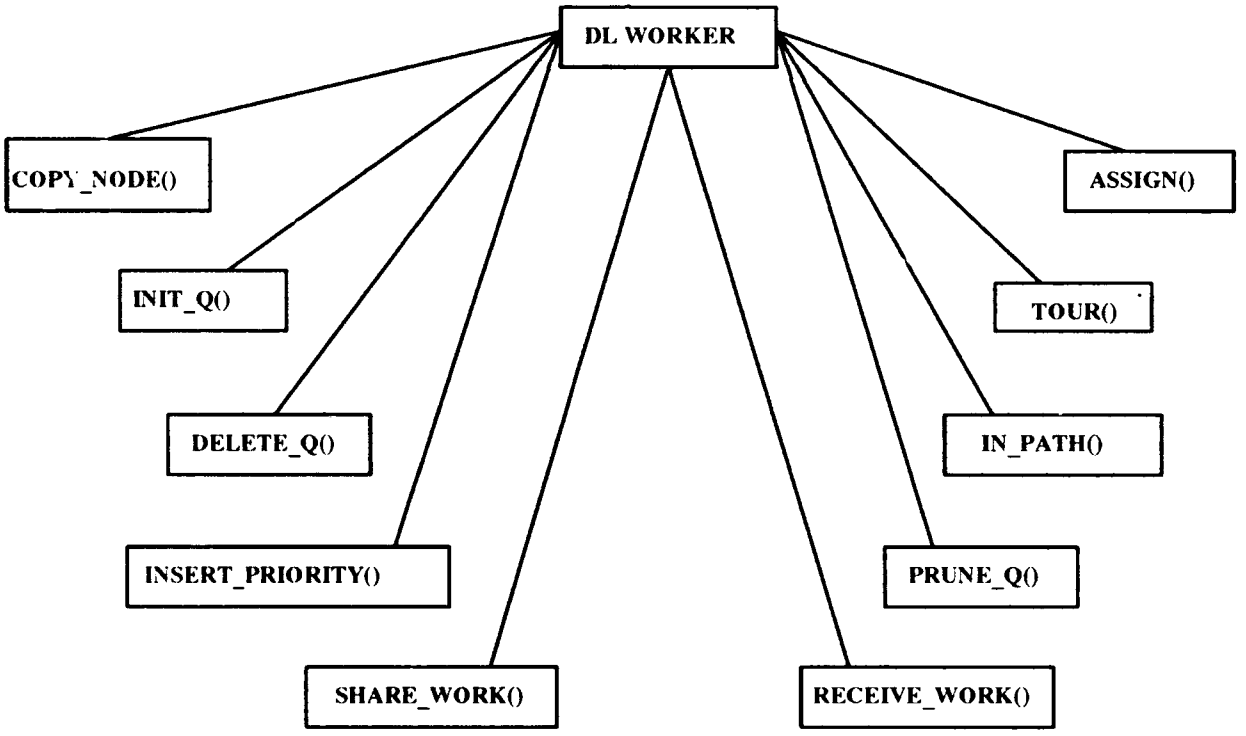


Figure A.5. Distributed List Worker Structure Chart

A.3.2 Distributed List with Load Balancing and Distribution The following structure charts are for the distributed list with load balancing and distribution algorithm:

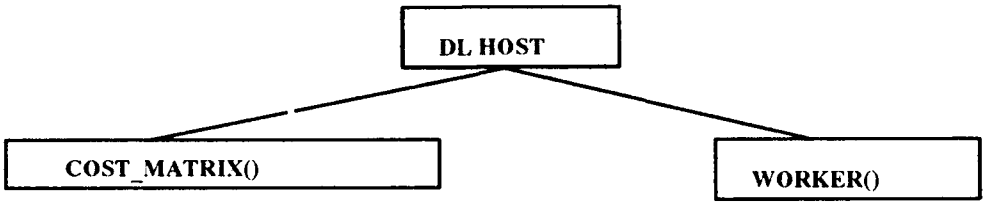


Figure A.6. Distributed List Host Structure Chart

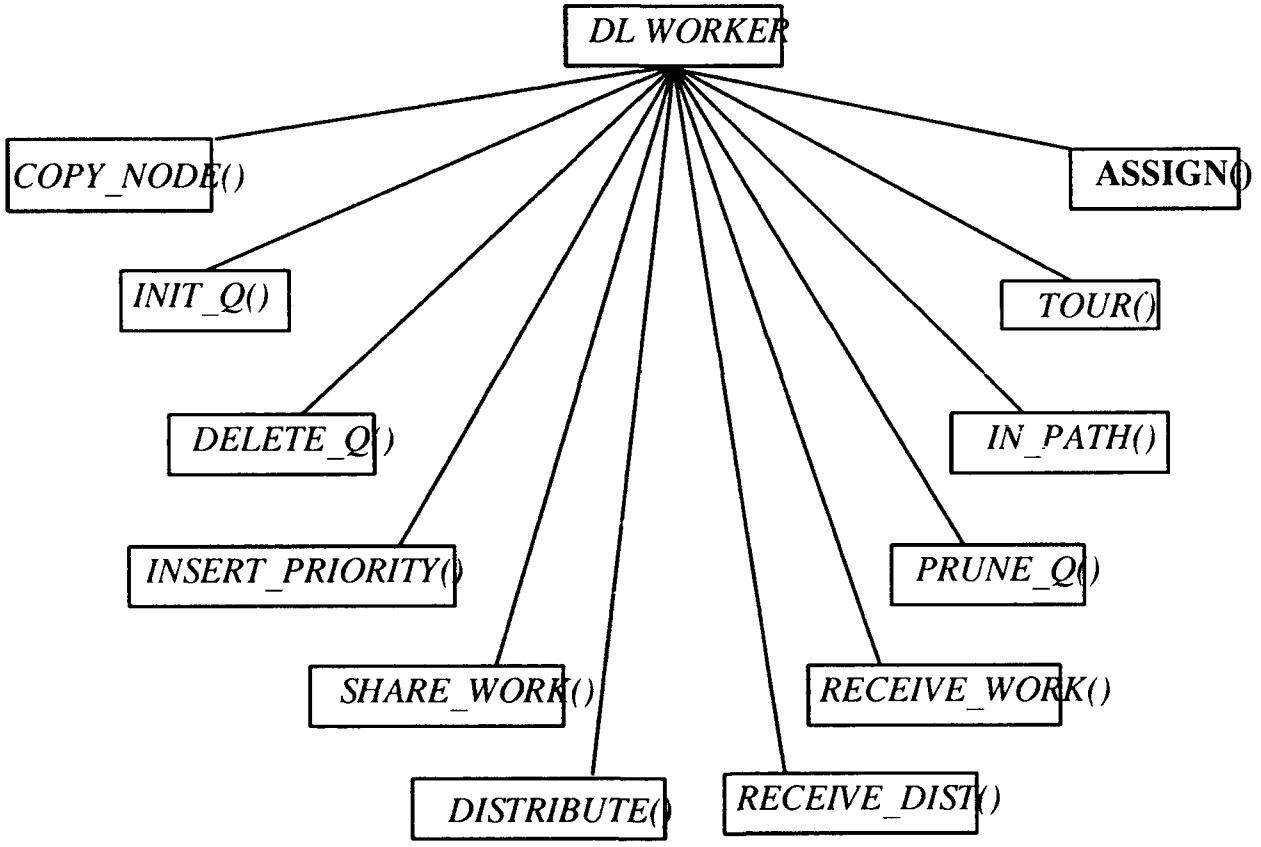


Figure A.7. Distributed List Worker Structure Chart

Appendix B. *Test Results and Data*

B.1 Introduction

This appendix presents the test data from all the algorithms tested. Each algorithm was tested using four different size problems; 22, 55, 65, and 100 cities and stored in files n22a, n55a, n65a, and n100a respectively. The description of the problems is in Appendix C.

The algorithms tested and their abbreviation used are:

1. Centralised list (CL) — Shown on charts and tables as tsp
2. Iterative Deepening A* — IDA*
3. Centralized list with levels — Shown on charts and tables as level
4. Centralized list with levels — level
5. Distributed list with no load balancing (DL_NLB) — Shown on charts and tables as nlb
6. Distributed list with load balancing (DL_LB) — Shown on charts and tables as dlsh
7. Distributed list with load balancing and distribution (DL_DIST) — Shown on charts and tables as dist

B.2 Data

This section presents the data in three different forms. The data is first presented in tables providing all the pertinent information collected about the algorithms. The data is then presented in graph form for ease of understanding and to display trends in the data. The CL algorithm was not run using 2 processors because then there would be the Control processor and only one Worker processor. This is the same as the sequential algorithm except with the parallel communication overhead. Also, some algorihms were not run using 32 processors due to hardware problems with the iPSC/2 computer.

FILE n22a				FILE n55a		
# NODES	RUN TIME	STATES EXPAND	AVERAGE EFFICIENCY	RUN TIME	STATES EXPAND	AVERAGE EFFICIENCY
1	16.83	25	0.823	754.57	148	0.996
2	NOT USED			NOT USED		
4	6.52	25	0.482	431.42	157	0.882
8	9.493	44	0.290	219.71	177	0.656
16	7.801	72	0.270	220.996	241	0.487
32	6.775	77	0.256	221.001	288	0.300

FILE n65a				FILE n100a		
# NODES	RUN TIME	STATES EXPAND	AVERAGE EFFICIENCY	RUN TIME	STATES EXPAND	AVERAGE EFFICIENCY
1	12341.3	1696	0.999	25724.1	737	0.999
2	NOT USED			NOT USED		
4	4153.2	1697	0.986	7201.7	581	0.938
8	1825.8	1710	0.981	2720.5	355	0.863
16	925.1	1767	0.973	2671.289	349	0.422
32	558.588	1963	0.949	2671.333	463	0.258

Table B.1. Centralized List Data

FILE n22a				FILE n55a		
# NODES	RUN TIME (sec)	NE	EFF	RUN TIME (sec)	NE	EFF
1						
2	29.8	74	0.856	1142.8	443	0.988
4	27.0	124	0.841	1142.8	739	0.988
8	27.0	224	0.842	1142.8	1331	0.988
16	24.0	419	0.829	1144.0	2501	0.987
32	24.8	803	0.832	1143.1	4115	0.988

NE = total states expanded

EFF = average efficiency per node

FILE n65a				FILE n100a		
# NODES	RUN TIME (sec)	NE	EFF	RUN TIME (sec)	NE	EFF
1						
2	22485	5017	0.838	41151	2810	0.947
4	17921	8866	0.843	33504	4103	0.950
8	15534	14971	0.851	31145	7113	0.952
16	13762	33025	0.825	28971	14892	0.966
32	13015	81132	0.861	30655	28463	0.967

NE = total states expanded

EFF = average efficiency per node

Table B.2. Distributed List with no Load Balancing Data

FILE n22a					FILE n55a			
# NODES	RUN TIME (sec)	TE	SH	EFF	RUN TIME (sec)	TE	SH	EFF
1								
2	25.1	74	7	0.805	1142	443	6	0.983
4	33.8	193	47	0.820	1140	783	74	0.993
8	24.1	305	235	0.884	1103	1495	243	0.709
16	23.8	517	469	0.798	1089	1752	701	0.738
32								

TE = total states expanded

SH = number of times work was shared

EFF = average efficiecy per node

FILE n65a					FILE n100a			
# NODES	RUN TIME (sec)	TE	SH	EFF	RUN TIME (sec)	TE	SH	EFF
1								
2	18846	3033	18	0.943	27221	1620	6	0.997
4	15291	7686	24	0.873	20117	3069	25	0.997
8	11529	14108	89	0.891	16381	6013	89	0.997
16	6394	26714	483	0.894	9210	11892	320	0.996
32	3229	33097	612	0.651	2995	24493	1184	0.996

TE = total states expanded

SH = number of times work was shared

EFF = average efficiecy per node

Table B.3. Distributed List with Load Balancing Data

FILE n22a					
# NODES	RUN TIME (sec)	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
1	not used				
2	32.7	92	0.805	33	12
4	27.2	157	0.784	39	35
8	14.7	252	0.576	198	292
16	13.2	424	0.694	94	400
32	not used				

FILE n55a					
# NODES	RUN TIME (sec)	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
1	not used				
2	1229	443	0.914	27	6
4	988	757	0.959	41	35
8	758	1210	0.833	72	281
16	297	2193	0.800	145	523
32	not used				

Table B.4. Distributed List with Load Balancing and Distribution 1 of 2

FILE n65a					
# NODES	RUN TIME (sec)	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
1	not uesd				
2	20011	4172	0.996	21	30
4	12977	7725	0.996	54	56
8	7034	14253	0.989	65	131
16	2529	21150	0.984	84	450
32	not uesd				

FILE n100a					
# NODES	RUN TIME (sec)	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
1	not uesd				
2	28712	1617	0.997	4	12
4	24619	3051	0.997	61	24
8	13011	5844	0.997	83	74
16	8901	11858	0.997	142	320
32	2935	20277	0.988	191	570

Table B.5. Distributed List with Load Balancing and Distribution 2 of 2

B.2.1 Execution Time Graphs This section presents the CL, DL_NLB, DL_LB, and DL_DIST algorithms' execution time data in graphical form for ease of understanding. The graphs show all four algorithms for each problem size.

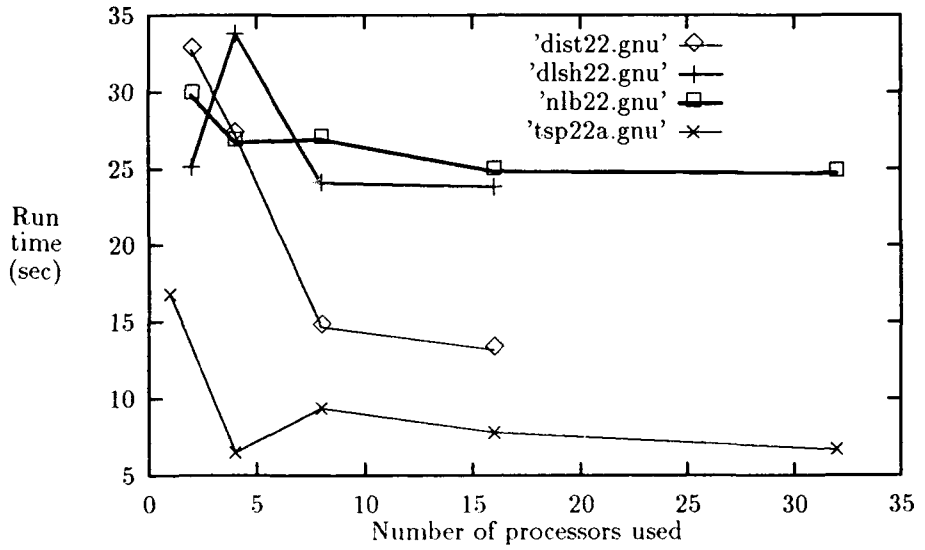


Figure B.1. Execution Time Data for 22 Cities

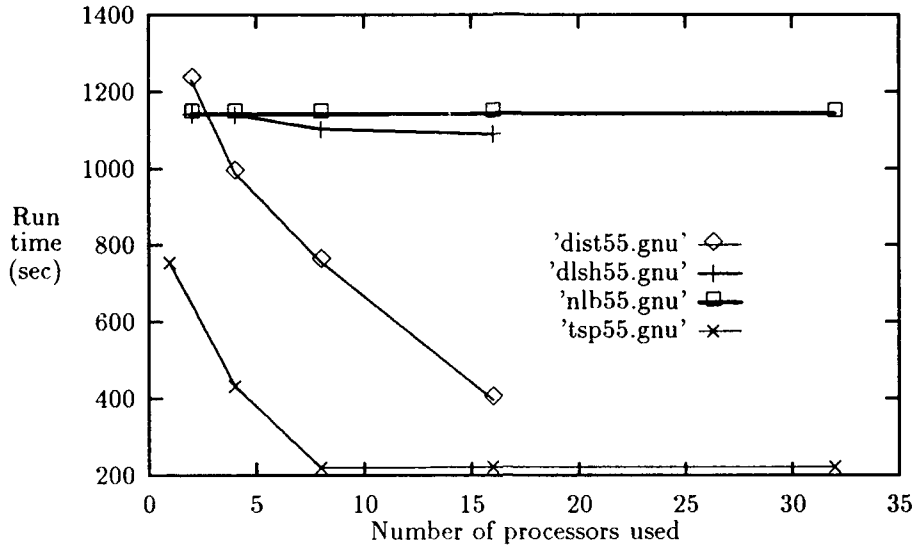


Figure B.2. Execution Time Data for 55 Cities

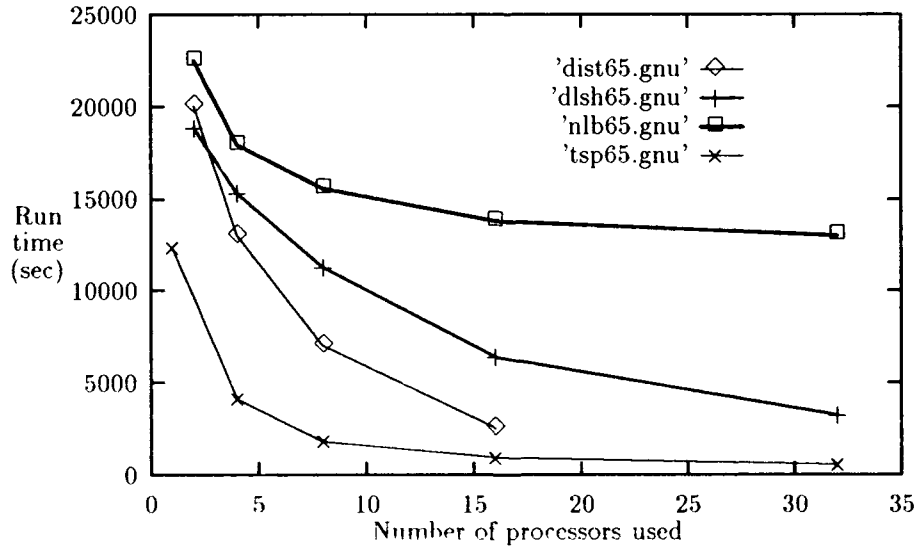


Figure B.3. Execution Time Data for 65 Cities

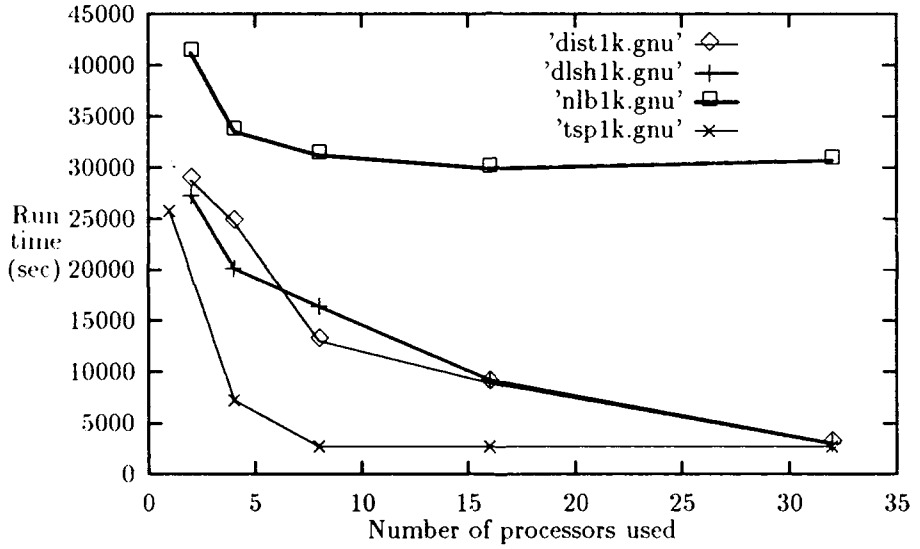


Figure B.4. Execution Time Data for 100 Cities

B.2.2 States Expanded Graph This section graphically presents the number of states expanded by all algorithms for each problem size.

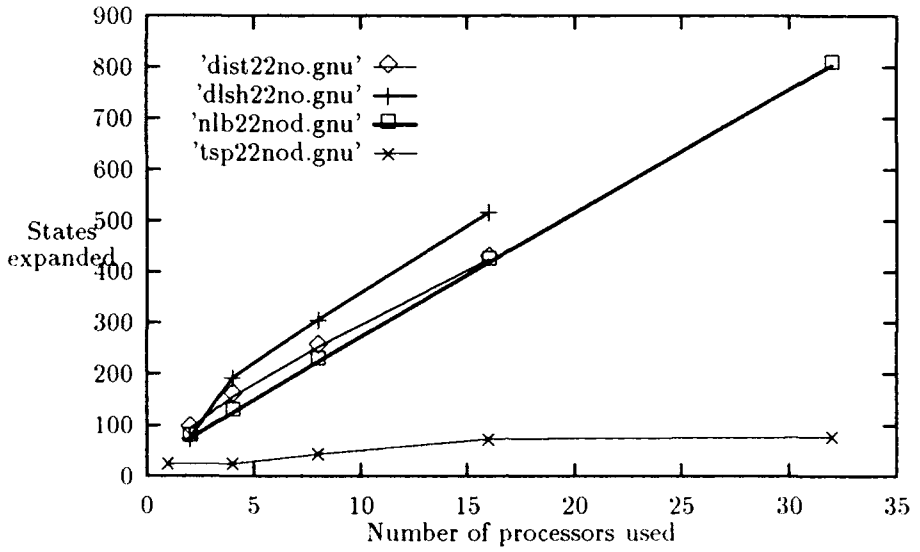


Figure B.5. States Expanded Data for 22 Cities

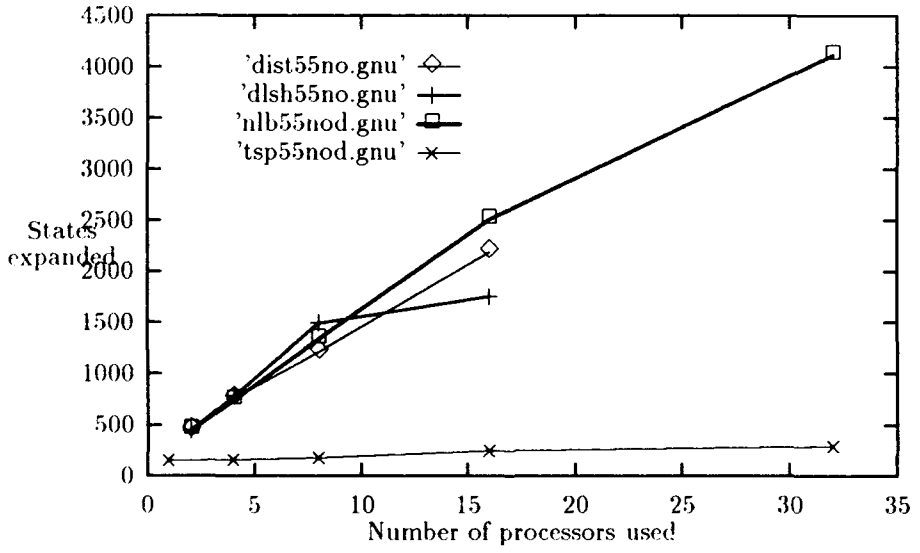


Figure B.6. States Expanded Data for 55 Cities

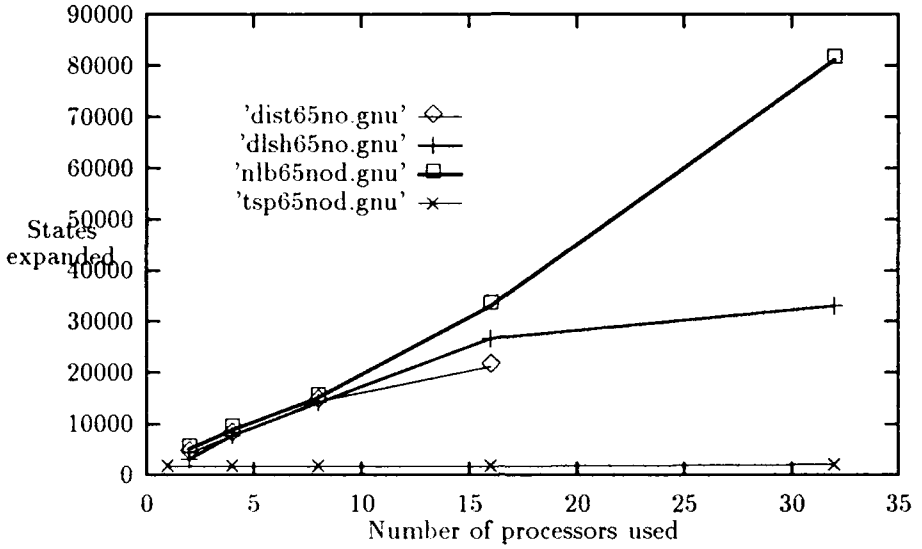


Figure B.7. States Expanded Data for 65 Cities

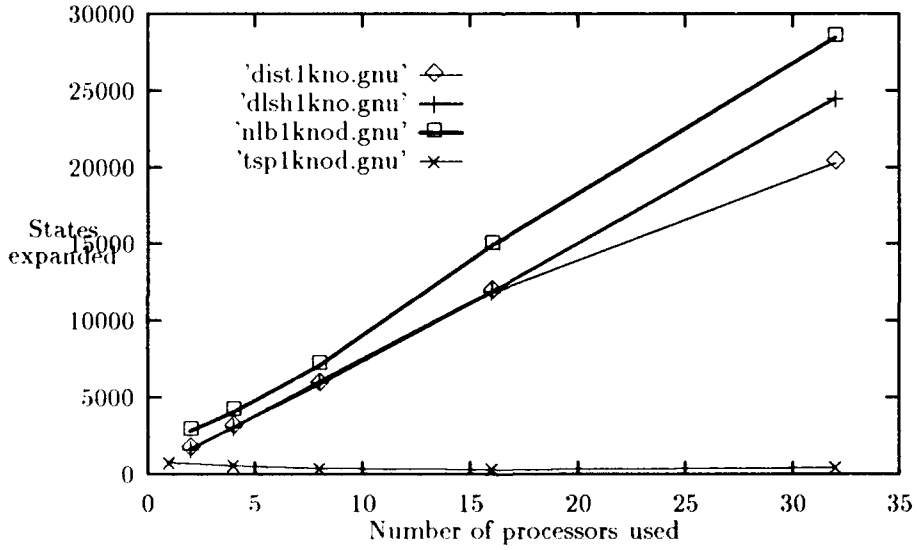


Figure B.8. States Expanded Data for 100 Cities

B.2.3 Share Data This section provides the data on load balancing . First, the table for the share variables and then the graphs are provided.

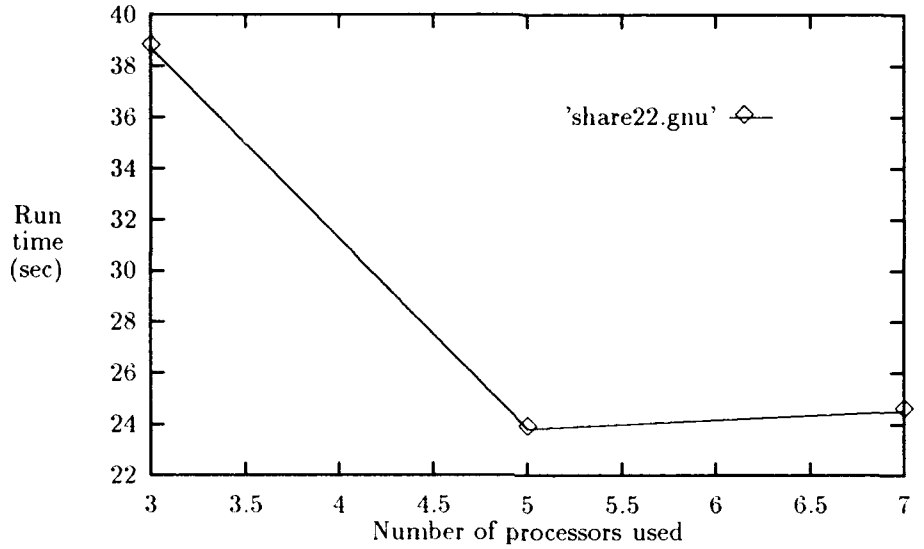


Figure B.9. Execution Timefor 22 Cities

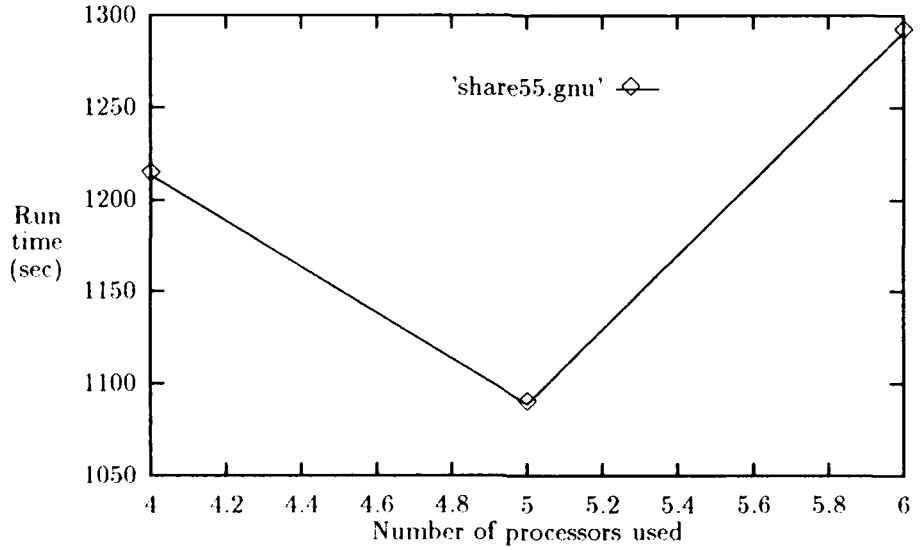


Figure B.10. Execution Time for 55 Cities

FILE n22a

N/S	RUN TIME (sec)	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
16/3	38.7	847	0.546	0	427
16/5	23.8	517	0.798	0	235
16/7	24.5	506	0.783	0	220

N/S/B = # of nodes used/share variable

DIST = # of times program distributed work
SHARE = # of times program shared work

FILE n55a

N/S	RUN TIME (sec)	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
16/4	1213	1740	0.607	0	1034
16/5	1089	1752	0.738	0	701
16/6	1291	1788	0.699	0	683

N/S/B = # of nodes used/share variable

DIST = # of times program distributed work
SHARE = # of times program shared work

Table B.6. Share Data 1 of 2

FILE n65a

N/S	RUN TIME (sec)	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
16/3	37295	34089	0.592	0	12382
16/4	6394	26714	0.894	0	483
16/5	6752	28113	0.884	0	494
16/7	6752	28113	0.884	0	494

N/S/B = # of nodes used/share variable

DIST = # of times program distributed work
SHARE = # of times program shared work

FILE n100a

N/S	RUN TIME (sec)	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
16/3	27838	18045	0.639	0	13056
16/4	9210	11861	0.997	0	320
16/5	9662	11804	0.997	0	316
16/7	9662	11804	0.997	0	316

N/S/B = # of nodes used/share variable

DIST = # of times program distributed work
SHARE = # of times program shared work

Table B.7. Share Data 2 of 2

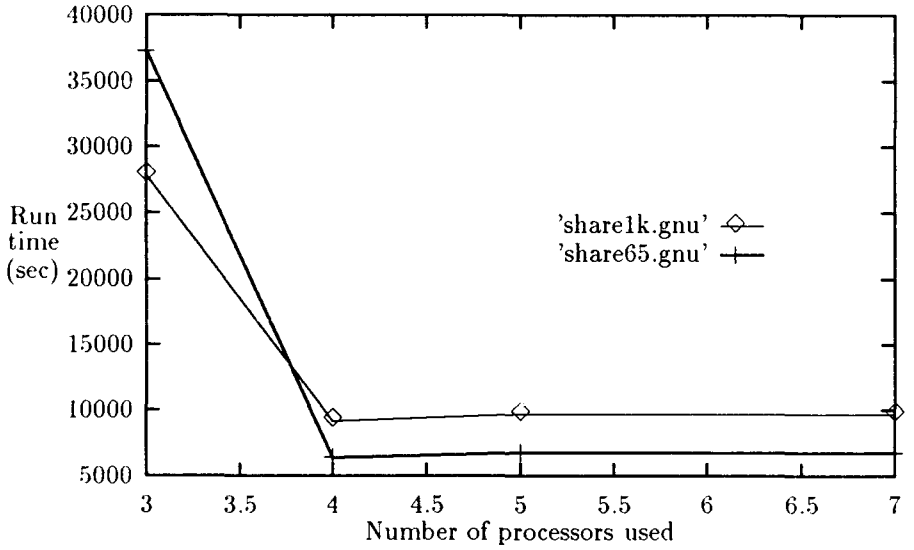


Figure B.11. Execution Time for 65 and 100 Cities

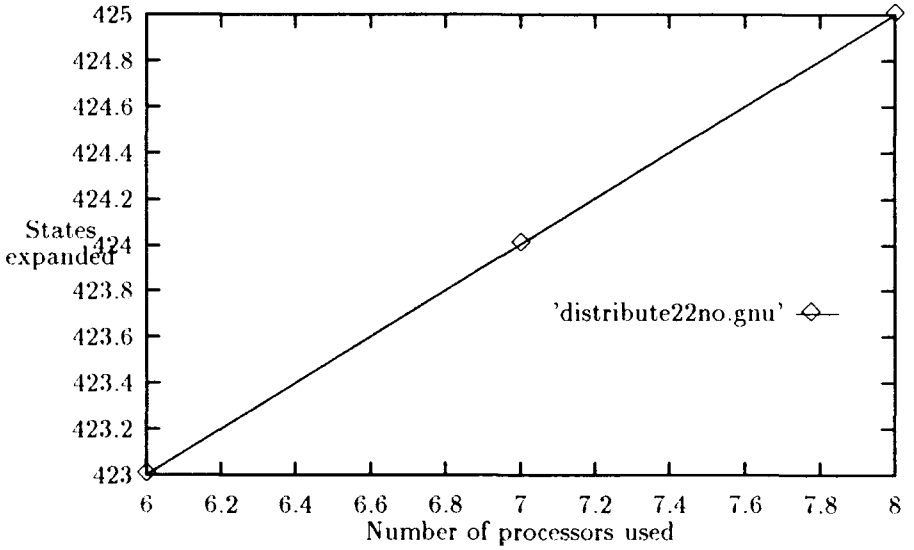


Figure B.12. States Expanded for 22 Cities

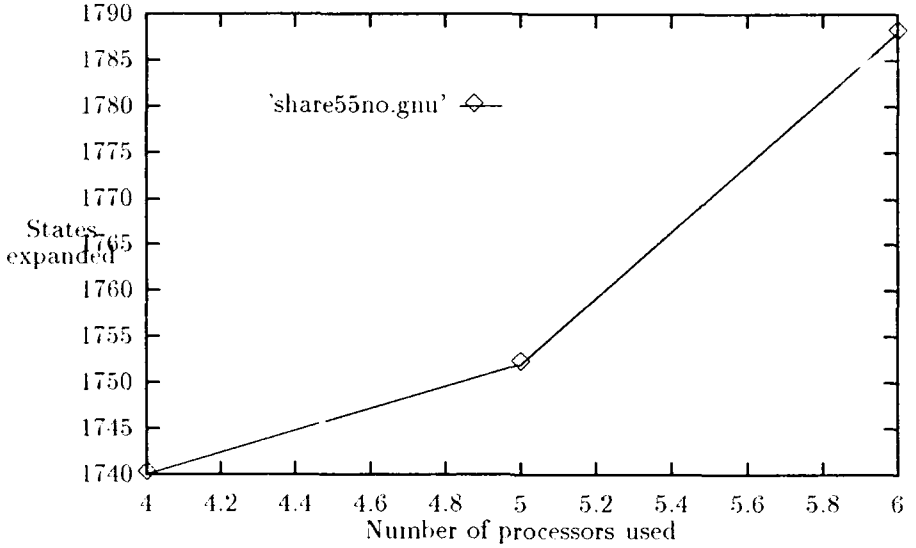


Figure B.13. States Expanded for 55 Cities

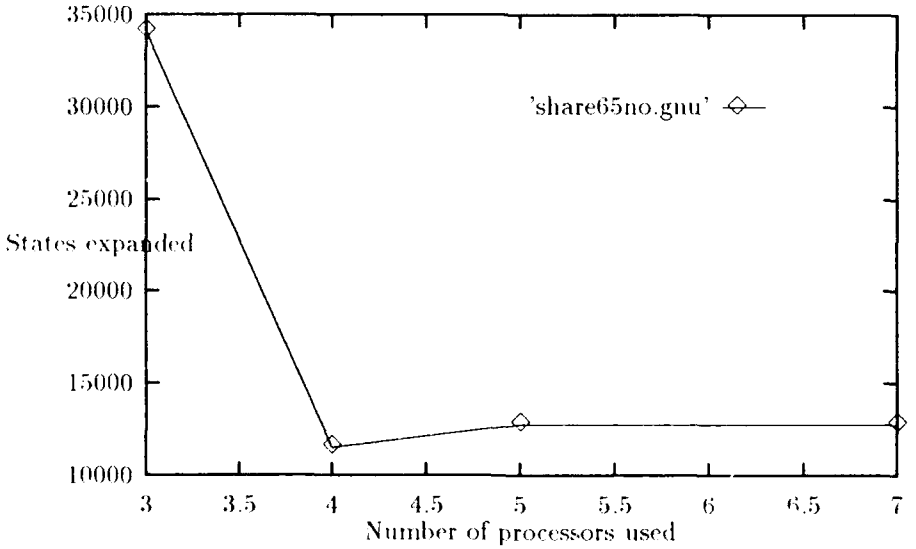


Figure B.14. States Expanded for 65 Cities

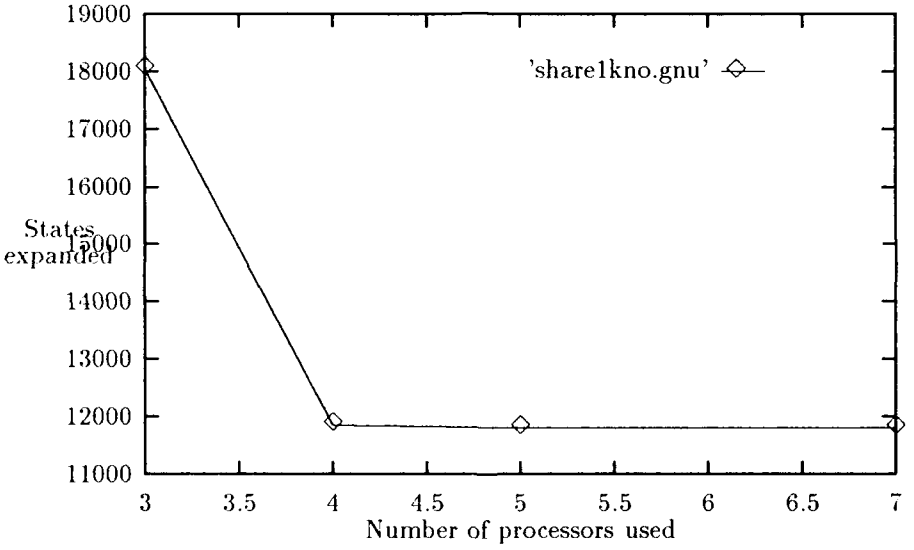


Figure B.15. States Expanded for 100 Cities

B.2.4 Distribution Data This section provides the data on distributing work . First, the table for the distribute variables and then the graphs are provided.

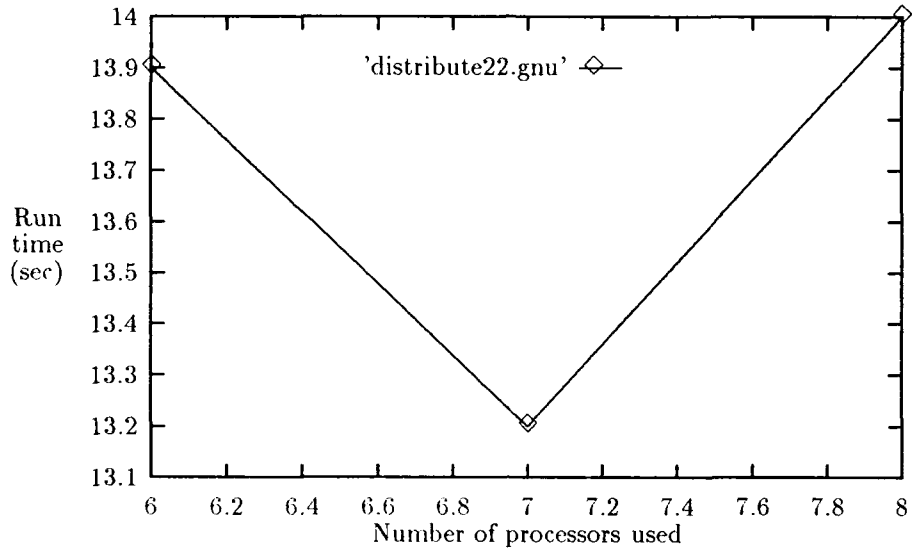


Figure B.16. Execution Time for 22 Cities

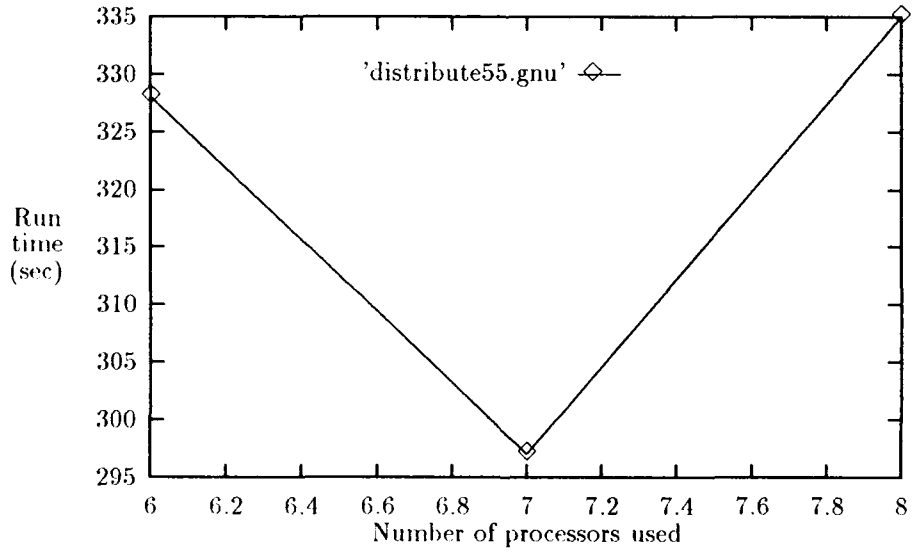


Figure B.17. Execution Time for 55 Cities

FILE n22a

N/S/D	RUN TIME	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
16/7/6	13.9	423	0.694	95	400
16/7/7	13.2	424	0.694	94	400
16/7/8	14.0	425	0.621	84	391

N/S/B = # of nodes used/share variable and distribute variable

DIST = # of times program distributed work

SHARE = # of times program shared work

FILE n55a

N/S/D	RUN TIME	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
16/5/6	328	2199	0.800	142	280
16/5/7	297	2193	0.800	145	281
16/5/8	335	2180	0.765	151	274

N/S/B = # of nodes used/share variable and distribute variable

DIST = # of times program distributed work

SHARE = # of times program shared work

Table B.8. Distribution Data 1 of 2

FILE n65a

N/S/D	RUN TIME	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
16/4/5	2852	20029	0.965	89	450
16/4/6	2529	21150	0.984	84	450
16/4/7	2997	26714	0.979	78	450

N/S/B = # of nodes used/share variable and distribute variable

DIST = # of times program distributed work

SHARE = # of times program shared work

FILE n100a

N/S/D	RUN TIME	STATES EXPAND	AVERAGE EFFICIENCY	DIST	SHARE
16/4/5	9672	11029	0.943	161	318
16/4/6	8901	11858	0.997	142	320
16/4/7	9513	12026	0.996	140	320

N/S/B = # of nodes used/share variable and distribute variable

DIST = # of times program distributed work

SHARE = # of times program shared work

Table B.9. Distribution Data 2 of 2

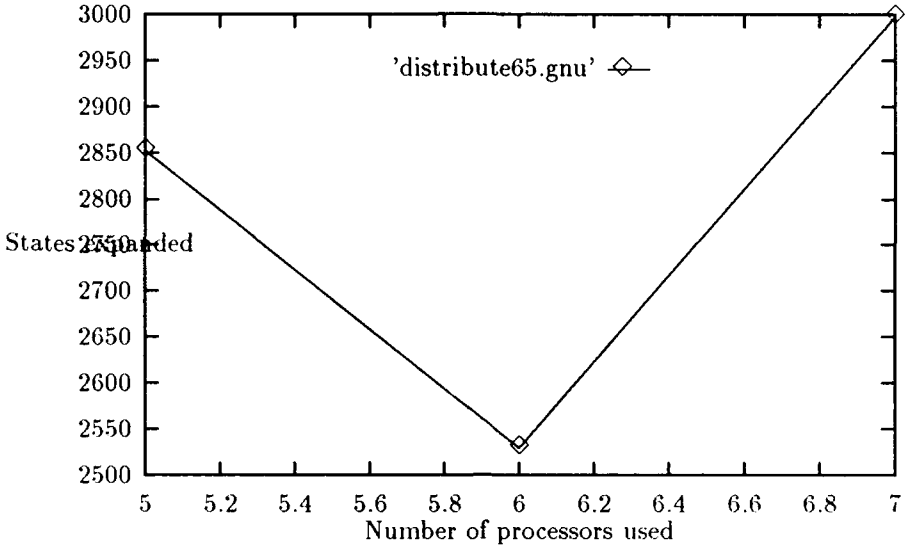


Figure B.18. Execution Time for 65 Cities

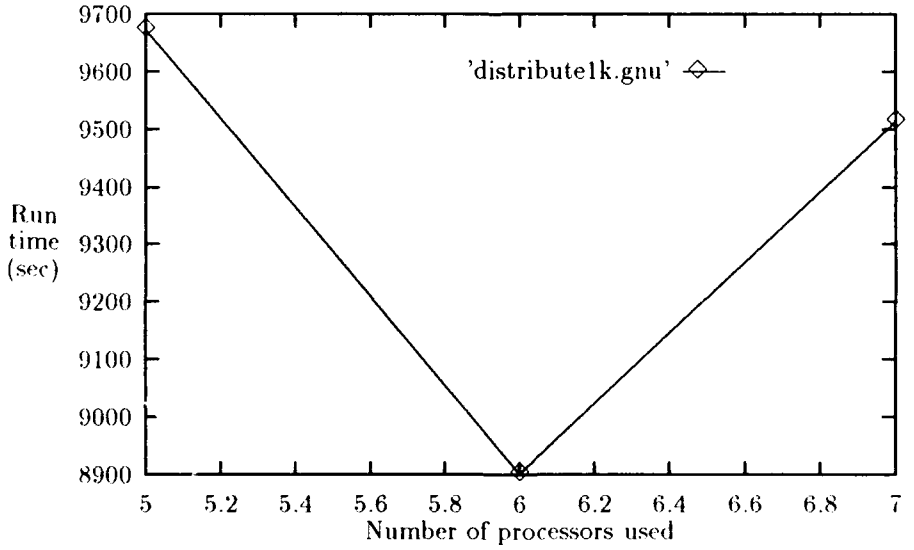


Figure B.19. Execution Time for 100 Cities

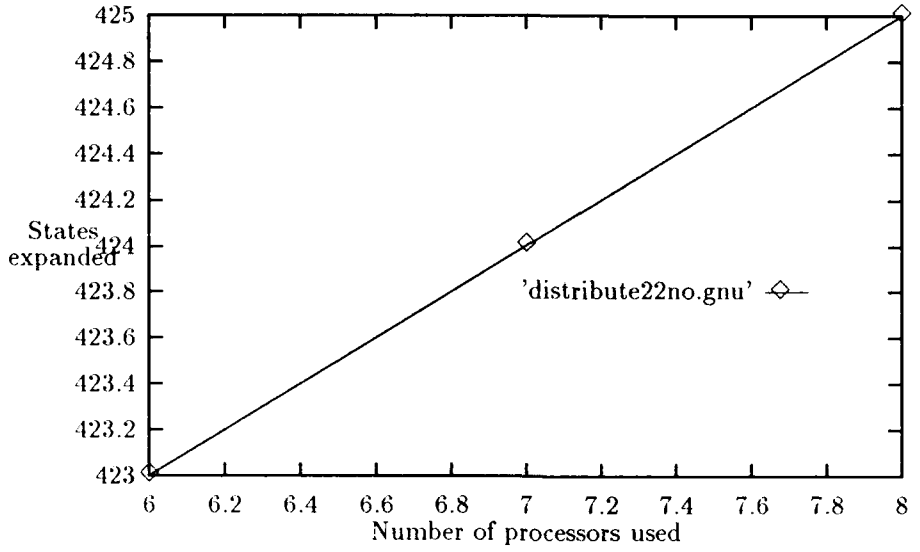


Figure B.20. States Expanded for 22 Cities

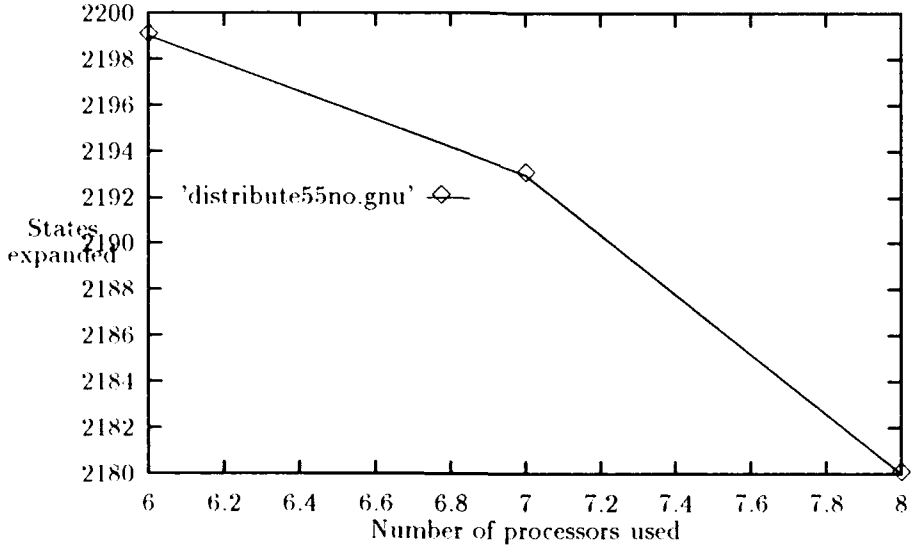


Figure B.21. States Expanded for 55 Cities

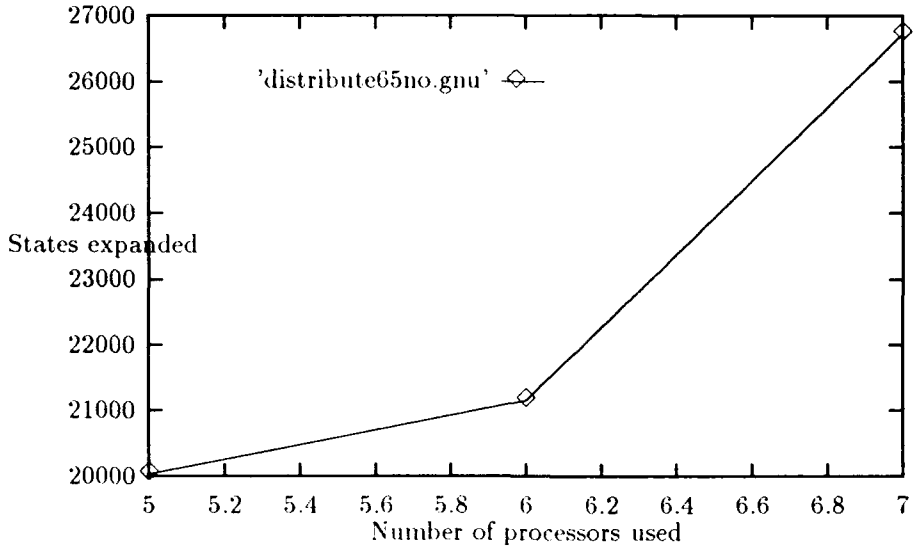


Figure B.22. States Expanded for 65 Cities

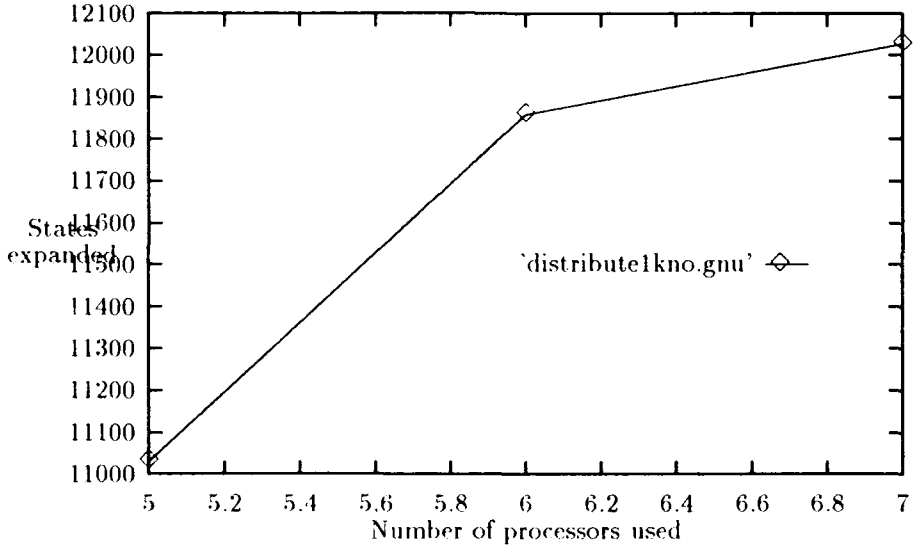


Figure B.23. States Expanded for 100 Cities

B.3 IDA Data*

This section presents the data comparing the centralized list algorithm against the IDA* algorithm. Both tables and graphs similar to those presented for the distributed list algorithms are presented here.

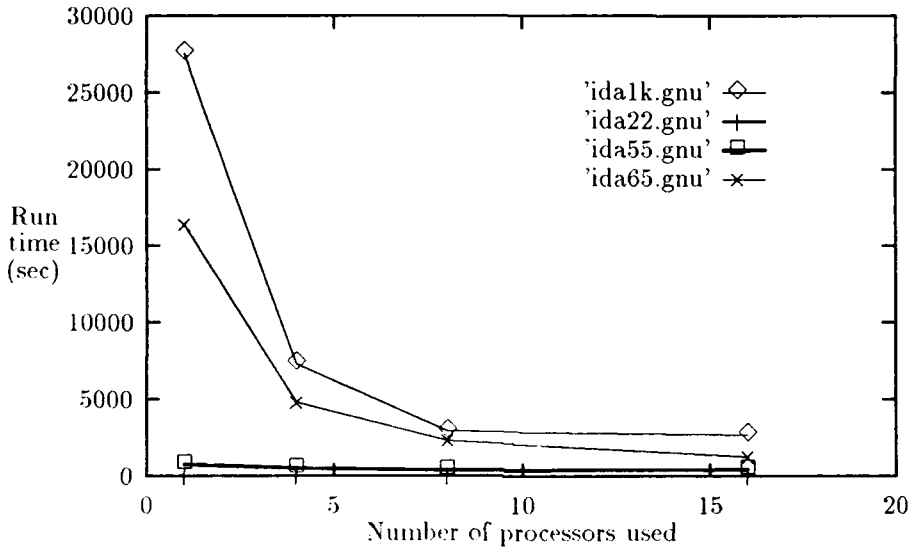


Figure B.24. IDA* Execution Time Data

FILE n22a					FILE n55a			
# NODES	RUN TIME	TE	IDA*	EFF	RUN TIME	TE	IDA*	EFF
1	15.0	25	19	0.850	744.6	148	139	0.997
2								
4	9.6	25	19	0.501	482.6	148	139	0.891
8	13.9	25	19	0.320	391.6	148	139	0.715
16	9.5	81	35	0.315	435.4	253	220	0.692
32	9.0	94	39	0.301	452.9	261	222	0.691

TE = total states expanded

IDA* = states expanded during IDA* DFS

EFF = average efficiecy per node

FILE n65a					FILE n100a			
# NODES	RUN TIME	TE	IDA*	EFF	RUN TIME	TE	IDA*	EFF
1	16392	1696	1682	0.999	27542	762	728	0.999
2								
4	4804	1945	1928	0.987	7311	345	339	0.941
8	2352	1979	1948	0.986	2915	366	349	0.806
16	1247	2009	1955	0.961	2722	381	351	0.471
32								

TE = total states expanded

IDA* = states expanded during IDA* DFS

EFF = average efficiecy per node

Table B.10. IDA* Data

FILE n22a (sec)					FILE n55a (sec)			
# NODES	RUN TIME	TE	LVL	EFF	RUN TIME	TE	LVL	EFF
1	17.0	25	25	0.866	761	148	148	0.998
2	not used				not used			
4	7.2	25	27	0.511	438	157	166	0.899
8	9.8	42	51	0.382	220	177	184	0.741
16	8.1	72	84	0.294	221	241	271	0.512
32								

TE = total states expanded

SH = number of times work was shared

EFF = average efficiecy per node

FILE n65a (sec)					FILE n100a (sec)			
# NODES	RUN TIME	TE	LVL	EFF	RUN TIME	TE	LVL	EFF
1	31057	1696	3762	0.999	26201	737	737	0.999
2	not used				not used			
4	27111	1697	5101	0.999	3821	581	341	0.966
8	22592	1710	7993	0.985	2650	355	329	0.894
16	20760	1767	11658	0.982	2687	349	361	0.500
32								

TE = states expanded using original TSP code

LVL = states expanded using levels

EFF = average efficiecy per node

Table B.11. Centralized List using Levels Data

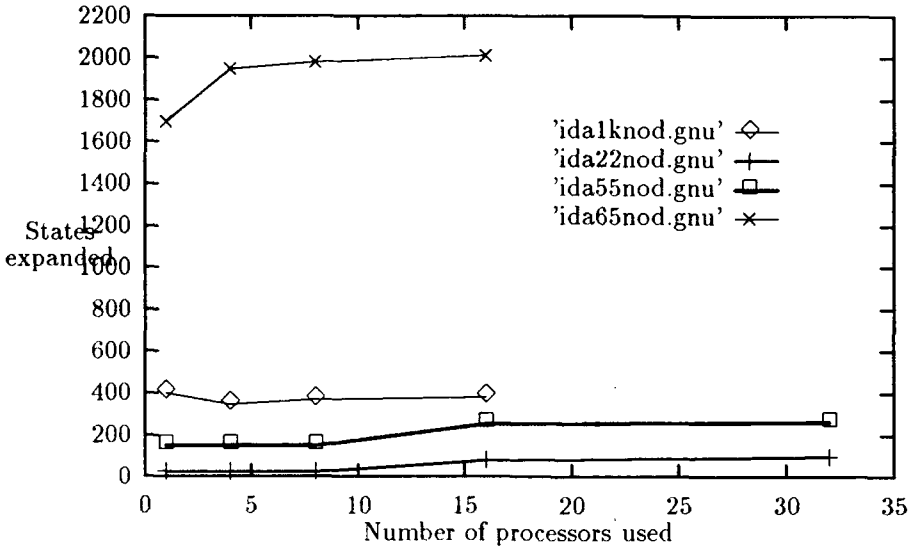


Figure B.25. IDA* States Expanded Data

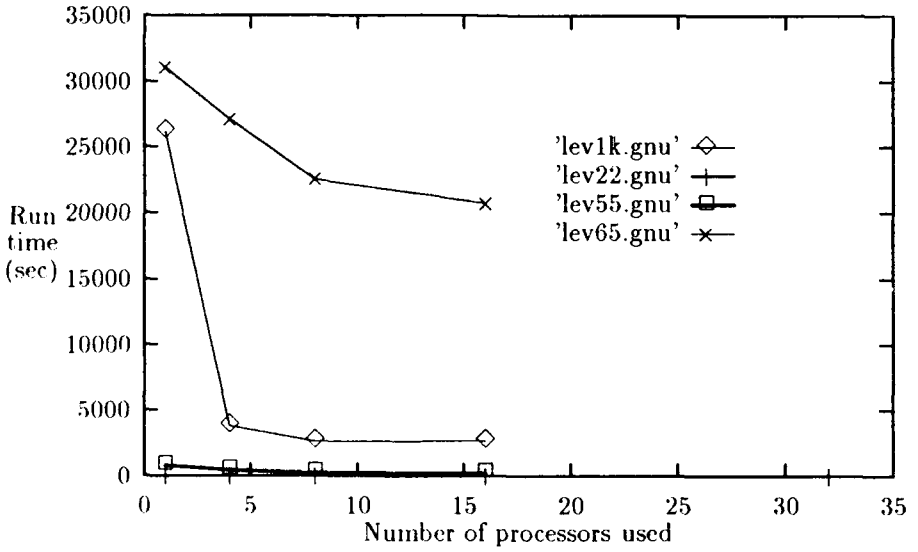


Figure B.26. Level Execution Time Data

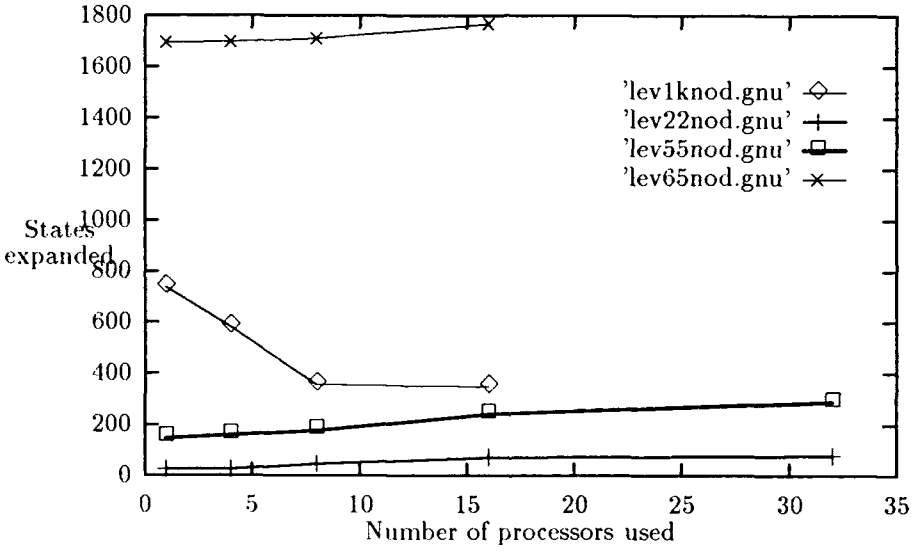


Figure B.27. Level States Expanded Data

Appendix C. *Problem Definition and Data*

C.1 Introduction

This appendix provides the cost matrices and solutions for the four main problems used in this research. Each section is divided into two parts: a cost matrix and a solution to the problem. As explained in Chapter II, the cost of traveling from one city to another is determined by selecting a city, finding that column, and then finding the intersecting row of the city that you are traveling to. For example, in the 22 city problem, the cost of traveling from city 4 to city 7 is 53 while the cost of traveling from city 7 to city 4 is 67. Again notice that the costs are not symetrical. A cost of 999 indicates infinite cost.

C.2 Problem n22a

This is the cost matrix for problem n22a:

999	63	63	64	93	31	82	32	82	92	91	92	40	70	73	81	58	55	99	97	43	96
55	999	17	95	80	37	53	57	36	43	86	19	80	31	86	99	74	82	44	53	61	91
39	78	999	73	11	63	88	76	34	54	68	40	62	96	84	78	60	77	44	51	3	86
49	11	48	999	50	73	53	74	43	26	0	23	93	81	78	43	63	98	62	11	71	60
72	46	38	62	999	65	82	42	40	93	53	56	44	42	59	14	37	13	55	90	1	3
89	57	33	18	97	999	69	52	50	70	32	3	29	42	36	31	30	23	63	34	40	14
0	79	95	67	87	60	999	44	4	9	41	61	28	44	58	79	54	32	4	14	85	71
17	98	6	78	82	39	40	999	90	88	26	57	85	90	97	16	25	54	68	18	52	4
78	9	85	23	7	46	1	47	999	4	35	98	37	59	79	47	98	94	53	96	7	69
7	72	41	85	15	28	8	88	12	999	2	0	22	6	90	6	1	20	88	45	71	61
69	37	76	84	28	24	37	84	61	7	999	15	3	25	83	93	12	79	16	33	19	40
6	74	59	81	93	81	44	1	35	39	9	999	94	35	41	30	33	33	83	60	16	80
96	6	31	2	35	13	70	51	17	58	9	87	999	48	11	58	82	55	67	79	80	72
59	23	25	63	8	1	51	81	70	70	58	62	91	999	79	0	58	69	15	29	82	93
18	37	66	2	75	75	56	60	59	98	70	22	53	45	999	41	31	41	35	89	8	79
85	59	18	21	47	50	57	0	42	19	49	71	41	19	9	999	26	90	2	40	73	35
10	14	99	73	63	58	89	33	69	70	14	81	13	0	36	33	999	3	96	62	21	76
25	36	75	8	11	56	13	3	46	16	64	67	84	86	99	5	37	999	76	54	55	15
93	74	82	0	38	44	81	99	16	48	25	46	76	79	67	21	20	81	999	54	67	49
50	25	47	54	20	31	97	67	96	21	39	9	67	69	99	32	58	57	52	999	59	6
22	15	56	81	74	50	91	71	75	73	95	15	8	24	91	76	62	21	54	88	999	11
7	45	78	7	92	93	44	69	71	35	22	20	56	64	73	40	46	85	97	39	91	999

This is a solution for the n22a problem:

1-6-12-8-20-22-4-11-13-15-21-2-3-5-18-10-17-14-16-19-9-7-1

at cost 180

C.3 Problem n55a

This is the cost matrix for problem n55a:

```
999  2  47   1   3  84  96  39  14  19  94  28  74  50  43  88  63  89  62  79  54  87
 66 74  58  25   8  76  43   8  29  28  38   9  37  78  66  37  57  57  59  27  93  20
 45 25  96  72   0   5  84  29  46  56  61

 87 999  1   1  51   6  93   3  67  62   4  80  39  58  93  68  84   4  43  41  58  79
 53 61  13  51  14   9  83  96  49   0  55  22  79  63  12  89  24  42  14  36  15   7
 72 79  90  23  85  72  88  41  25  91   2

 87   3 999  32  12   2  13  17  75  16  59  73  26  47  39  87  93  28  65  44  74   4
 56 38  71  78  14  65  44  88  89  62   4  70  82   1  94  39  90  52  72  95  33  49
  4 42  79  90  63  23  29  27  73  64  45

 95 19  12 999  69  94  46  29  47  29  72  74  86  17  80  25  63  57  25  34  55   3
 92 52   9  76  66  27  30  34  29  37  81  98  90  69  67  49  50  61  23  21  92  68
 26 84  78  15  12  76  60  96  75  89  59

 32 69  90  74 999   9  72  19  44  86   9  59  12  66  99  25  72  31   5  88  90  20
 95 45  89  50  45  85  75  78  62  12  59  90  94  96  99   4   2  19  76  60  36  46
 85 82  74  29  45  40  80  79  29  65   0

 90 28  53  65  80 999   5  47  95  16  73  51  85  55  58  80  53  74  40  26  99  87
 14 34  80  79  60  71  61  66  77   7  63  17  39  68  77  21  90  67  71  57  38  59
 74 85  37  79   1  40   4   1  66  38  74

 37 52  65  92  25  63 999  49  42  51  50   0  50  20  13  92  79   7  98  63   5   4
 88 99  98  34  69   9  24  56   1  78  17  12   4  57  80  21  60  16   5  68  41  21
 81 88  69  69  86   2  64  74  90  32  84

 65 72  55  30  53  48  75 999  31  49  27  10  98  65   9  63  25  87  80  66  66  74
  5 33  36  55  37  21  36  73  90   7  18  82  28  13  67  51  77  31  81  82  64  83
 42 42  24  72  29   6  25  86  31  57  84

  2 38  80  59  35  99  18  59 999   8  73  40  76  53  46  97  66  12  39  68  93  52
 93 20  58  87  38  73  35   4  56  19  68  60  97  66  52  49  45  44  11   0  67  22
 30 82  77  86  95  89  24  29  16   8  19

 36 66  65  49  51  79  95  47  12 999  64  54  27  59  56  34  53  27   9  29  41  68
 77 45  53  29  96  22  85  39  43  40  36  72  24  45  54  62  23   3  82  32  33  84
  9  1  48  98  76  54  41  98  74  13   5

 85 77  76  25  52  66   3  85  71   7 999  21  96  87  12  89  49   9  89  19  89  67
```

22 25 43 50 54 55 15 57 36 19 46 15 70 89 59 38 10 50 63 94 9 47
70 68 44 67 4 24 82 89 74 6 72

92 96 66 66 0 32 74 64 36 68 49 999 64 38 65 24 22 70 63 68 98 30
55 16 66 34 76 29 79 66 11 86 97 16 34 52 45 38 74 2 90 14 44 2
70 33 27 71 74 76 19 54 10 85 81

74 94 80 57 7 17 18 84 11 65 46 54 999 59 35 94 19 12 45 23 53 56
50 99 25 27 97 44 94 46 96 46 66 38 39 90 79 39 11 89 29 35 73 50
93 60 23 98 62 92 54 60 79 59 41

1 55 87 88 74 64 75 52 80 53 25 41 76 999 79 76 95 62 38 64 85 60
60 37 82 63 23 47 41 18 10 32 98 45 38 31 7 26 37 71 92 71 71 30
74 75 96 2 6 3 60 90 57 32 2

6 [illegible] 73 75 5 93 79 74 58 87 48 48 28 78 999 52 33 30 70 6 70 38
72 [illegible] 14 38 7 83 98 52 12 14 71 14 97 24 80 68 27 20 4 89 89 38
83 93 88 64 37 21 59 19 92 80 68

55 81 3 64 81 8 10 3 14 91 9 60 84 56 67 999 23 9 39 83 96 15
94 27 54 34 12 75 42 20 64 76 90 41 57 86 30 55 70 73 46 99 58 12
46 69 42 43 92 42 87 25 83 59 90

49 23 57 54 28 21 27 70 95 65 15 71 42 1 39 20 999 36 85 31 41 6
90 35 34 59 43 39 64 41 39 58 65 63 41 59 35 81 69 66 82 25 82 45
89 28 88 74 15 12 31 97 79 44 81

14 81 24 80 48 58 18 96 76 5 22 82 55 31 74 49 74 999 29 0 28 23
53 67 84 81 34 28 56 72 83 13 83 14 53 72 70 41 25 49 37 16 16 66
82 52 55 74 31 67 25 79 53 36 6

87 25 18 8 69 66 66 40 2 8 18 80 15 46 17 98 61 23 999 32 92 49
47 88 61 21 36 98 76 67 35 30 32 45 30 58 64 30 91 73 67 66 47 56
9 52 36 2 80 47 62 47 85 82 39

89 7 56 28 33 41 22 86 76 52 5 64 45 43 28 64 87 61 30 999 9 63
12 67 32 47 11 2 41 28 33 86 54 41 56 18 54 13 68 20 48 37 61 94
39 39 46 97 76 54 87 16 78 18 45

93 57 63 3 87 33 89 62 40 54 58 9 84 39 22 93 52 55 39 90 999 40
15 68 94 24 89 19 84 87 21 10 60 17 96 52 77 0 50 80 54 69 57 45
8 97 59 20 45 98 43 96 77 44 63

76 98 75 53 50 28 33 19 4 90 40 64 44 19 40 24 98 15 87 51 34 999
31 65 97 74 35 19 78 18 66 1 4 79 22 6 88 43 31 50 32 6 62 17
11 60 95 5 62 16 4 22 54 49 61

1 91 34 31 27 86 7 35 11 1 4 86 54 75 51 18 66 32 33 88 20 43
999 85 4 27 21 48 70 21 76 74 90 53 18 51 61 52 92 56 74 43 3 89
66 42 26 20 75 79 42 46 4 70 68

89 18 75 4 19 47 48 30 53 42 99 41 57 1 45 19 22 5 97 70 45 95
10 999 49 22 4 18 27 11 35 11 24 73 82 84 28 29 29 52 87 54 53 22
66 91 88 3 84 47 94 76 42 59 24

83 15 91 40 17 2 9 23 98 77 65 78 90 57 54 95 68 20 5 28 3 14
39 76 999 76 71 40 8 84 70 97 93 69 29 86 87 8 96 92 6 7 23 49
60 31 35 46 49 13 29 62 10 93 88

55 47 68 92 33 56 58 59 33 34 63 55 23 62 57 85 92 71 99 60 37 23
12 85 1 999 79 22 78 5 47 77 1 44 0 62 22 82 48 37 53 49 63 73
89 46 47 3 21 81 98 85 42 33 31

7 12 23 66 15 26 73 59 73 49 51 23 62 2 39 38 29 15 3 89 64 58
58 0 11 7 999 6 51 61 26 1 88 87 0 60 44 29 34 0 6 69 8 94
61 57 0 79 13 7 18 32 75 15 50

21 48 56 92 29 55 40 27 97 3 97 80 99 71 88 90 44 53 83 33 2 12
18 83 8 69 47 999 50 48 1 22 87 57 59 71 71 49 17 54 30 5 22 76
97 10 22 30 77 91 34 42 93 51 66

70 48 53 38 60 14 4 43 2 51 16 37 66 22 77 5 18 79 64 67 5 43
32 33 46 12 83 27 999 32 36 54 98 84 55 34 61 22 59 62 93 87 91 0
27 81 31 99 3 64 50 44 93 59 57

84 93 77 39 26 43 45 93 1 40 89 63 34 12 78 45 95 35 28 82 14 80
55 2 77 91 77 14 39 999 64 19 56 60 76 17 69 82 67 5 41 93 13 60
57 27 77 75 47 87 42 93 66 15 79

32 21 85 97 92 40 67 49 63 8 62 52 10 39 72 57 74 68 41 23 89 89
43 7 62 90 94 5 35 81 999 21 22 21 14 69 10 30 63 10 87 42 25 72
72 83 21 12 7 77 45 31 4 25 74

44 62 50 42 83 40 57 59 45 5 69 94 87 69 26 76 38 57 70 98 44 12
17 46 10 20 44 89 58 39 27 999 89 12 7 60 87 68 51 48 1 23 30 85
16 5 6 36 43 49 9 95 36 43 40

6 11 98 68 70 64 94 4 12 10 28 12 39 20 64 31 47 35 19 75 52 85
0 37 1 84 68 66 38 22 9 73 999 1 76 72 27 20 14 50 27 43 41 28
51 71 18 2 26 11 53 57 51 14 85

24 38 77 54 11 57 82 25 94 54 0 65 16 81 57 62 88 29 37 85 40 39
13 36 88 6 79 42 17 46 56 47 37 999 38 71 2 23 44 47 93 22 78 51
65 42 87 14 81 3 13 61 7 57 60

80 16 69 62 79 55 75 16 94 17 79 88 75 44 70 69 23 18 6. 0 58 93
22 36 26 54 38 10 31 2 57 38 65 36 999 0 74 88 8 30 57 11 84 62
60 70 23 49 7 27 7 67 93 7 64

52 59 26 14 46 3 52 31 17 38 87 64 82 10 65 81 80 6 59 87 42 14
79 81 79 6 53 49 48 18 31 56 23 31 49 999 5 88 97 44 82 36 91 3
9 92 95 23 49 8 65 91 12 19 2

50 87 42 54 56 12 72 96 20 75 17 57 67 87 66 53 10 43 76 5 72 32
48 40 25 52 93 84 94 92 33 29 80 12 77 44 999 61 75 47 98 45 93 76
87 17 31 61 88 6 29 88 24 18 66

43 45 20 60 9 10 56 3 3 53 17 86 84 3 95 67 20 52 31 48 70 71
71 90 61 5 73 53 52 32 70 34 49 34 71 75 50 999 0 67 40 23 92 92
38 55 70 19 64 85 42 50 70 89 3

67 96 66 32 66 95 71 49 20 98 78 83 62 58 78 80 84 8 95 49 21 59
5 19 9[illegible] 21 8 72 60 19 59 46 84 28 17 49 29 58 999 79 72 63 70 22
92 26 36 39 84 97 27 1 85 73 6

80 30 10 8 3 39 39 57 54 69 91 47 95 60 55 83 62 64 63 12 77 50
83 71 10 83 33 96 59 10 99 31 50 53 6 70 52 90 47 999 6 83 10 2
93 63 58 56 70 74 97 84 94 19 0

99 45 27 85 53 28 3 98 4 18 45 46 8 74 89 28 27 50 10 83 93 17
56 89 91 82 89 47 92 69 32 74 33 46 51 38 48 19 71 72 999 71 10 19
4 51 94 16 16 43 75 59 70 75 56

54 23 8 48 41 94 16 36 76 92 18 87 36 99 51 40 16 12 9 88 53 27
99 19 50 99 76 59 14 98 28 22 9 51 36 60 24 59 94 77 86 999 31 12
48 95 22 89 80 62 54 72 88 24 87
21 20 47 46 58 47 51 24 65 43 20 86 26 74 27 33 26 49 44 85 23 37
68 73 95 39 46 54 35 49 62 64 52 47 44 86 35 10 74 49 69 26 999 93
94 75 5 30 80 46 27 69 44 6 57

86 55 0 65 78 25 88 4 28 39 96 30 79 98 42 22 11 78 21 92 78 1
66 6 37 36 29 65 60 47 73 81 43 40 68 53 79 2 75 85 74 98 22 999
45 41 82 64 80 82 21 72 8 69 18

35 20 48 73 41 96 23 58 49 5 22 77 4 44 81 73 89 51 44 39 38 17
89 30 35 93 35 60 7 91 64 77 35 25 87 72 58 68 8 30 78 61 99 48
999 7 25 87 50 41 27 78 57 47 18

65 23 11 93 9 51 45 37 77 70 69 27 38 15 84 8 8 65 22 72 45 11
14 66 83 33 60 1 72 57 57 32 80 91 59 35 6 16 70 18 26 38 58 26
77 999 28 96 25 24 97 96 62 22 93

28 17 92 43 93 25 81 45 45 54 90 96 71 29 16 88 39 25 8 83 41 91
76 38 34 39 2 9 29 53 84 62 90 70 14 58 4 46 23 7 18 33 63 13
52 26 999 57 4 7 27 63 91 75 86

95 78 77 53 27 37 3 27 13 90 35 70 28 76 42 45 5 90 21 25 38 42
25 11 7 3 31 72 39 5 99 71 21 18 65 48 31 45 18 65 83 22 7 41
20 84 74 999 61 94 22 65 70 37 40

67 60 4 1 34 23 60 25 74 69 4 72 15 52 86 82 25 42 28 4 11 83
52 27 17 35 20 99 62 19 50 81 37 16 42 14 41 25 94 3 75 2 50 86
70 9 8 86 999 24 36 6 26 27 48

```
98  99   6   3   2  21  25  36  83  35  12  20  78  67  79  66  85  48  30  19  25  82
11  74  53  83  36  81   2  15  93  31  68  90  62  97  83  29  38  71  14  53  19  40
60  23   9  49  17 999  91  84   3  55  20

21  15  57  56  64  13  28  36  24   8  91  38  20  12  72   2  48  46   0  79  70   7
96  13  25  59  86  52   1  64  57  30  49  50  63  41  97  65  88  55  97  73  64   9
37  77  66  56   3  38 999  37  46  66  55

86  47  90  60  96   3  91  11  12  82  78  41  57  99   6  29  46  86  26  60  97  87
84  48  87  47  24  63   2   4  19  41  90   1   2  95  62  40  56  15  54  18  92  65
 2   8  15  33  90  71  59 999  22  43  46

53  44  88  71  62  24  77  63  59  24  60  89  25  90  42  52  12  13  48  84  80  46
69   6  33  64   4   0  11  22  62  84  85  82  45  93   1  98  87  54  41  22  76  37
24  [illegible] 66  41  58  69  82  68 999  93  87

72  16  84   8  28  97  98   5  26  47  71  78  58  16  66  48   4  61  62  88  74  35
34  18  12  37  79  47  72  24  20   5  69  86  79  14  30  20  37  95  47  45  24  75
91  51   9  16   4  92  44   1  86 999  17

55  68  30  91  10  38  26  49  16  31  26  85  46  61  37  75  47  32  17  38   3  46
64  49  21  52  75  43  55  75  67  19  24  77  50  75  39  99  76  12   6  43  17  67
62  59   3   0  49  67   0  76  89  99 999
```

A solution to the 55 city problem is:

```
1-2-18-20-28-31-53-37-50-29-7-12-40-44-3-36-6-49-4-22-32

-10-46-16-8-15-41-45-13-5-55-51-19-9-42-33-23-43-47-27-35

-30-24-48-26-25-21-38-39-52-34-11-54-17-14-1
```

at cost 134

C.4 Problem n65a

This is the cost matrix for problem n65a:

```
999 46  65  61  31  89  45  39  35  32  86  72  44  40  32  64  16   2  97  66  92  82
```

```
 8  95  65  62  79  14  60  19  49  20  70   8  27  22  66  17  63  35   1  30  74  32
86  33  42  58  65  62  30  26  69  21  81  31  29  48  53   0  39  41   1  53  20

11 999  25  38  91  72  77  86  88  95  32  29  88  22   9  96  45  69  16  89  40  90
49  98  74  49  26  83  10  84  21  11  72  73  20  15  38  22  86   7  40  37  53  32
10  24  33  43  32  37  18  29  32  95  97  52  79  87  13  12  46  16  16  81  56

 4  84 999  55  31   7  90  92  13  90  12   5  72  28  87  75  46  69  57  75  35  46
40  93  74  76  83  98  77  13  88  96  14   0  80  22  53  75  90  82   3  13  70   7
50  93  43  51  15  90  68  62  96  57  95  36  25  15  87  87  28  19   9  71  28

79  16  82 999  18  43  53   0  82  14  70  63  84  51  72  40  11  42  35  48  55  51
49  10   9  60  29  49  82  19  87  36  56  46  57  93  93  34  84  80  34  61   9  29
70   6  82  62  46  31  99  94  43  21  22  22  15  56  66  20  85   4  63  29  35

66  14  44  41 999  44  86  71  15  43  66  89  64  99  50  12  12  44  80  20  54  61
35  98  37  65  33  70  72  66  13  70  42  22  52  22  84   5  64  69  28  35  74  90
81  13  15   3  72  25  85  45  38   9  75  36  16  13  23  25  71  73  80  33  99

86  86  44  36  57 999  89  49  83   5  79  13  31  63  58  95   4  10  38  50  41  40
35  20   0   7  75  81  37  98  72  28  67  86  48  58  70  77  69  68  67   5  16  65
49  12  54  91  58  51  80  47  65  34  70  80  13  47  49  28  52  15  38  52  86

71  23  40  55  47  13 999  62   7  11   3  51  94  85  15  76  28  29  70  47  39  75
73  26  23  86  54  20  54  10  57  80  72  48  46  97  49  89  96  14  44  55  72  43
20  19  58  18  84  74  20  87  87  56   3  14   6  30  18  41  13  34  74  37  17

40  70   2  69  62  27  53 999  59  58  79   5   5  55  95  29   8   8  91  71  90  35
20  14  14  13  52  41  51  16  40  55  20  65  98  54  95  98  54  91  74  59  62  32
91   4  41   0  18   7  63  98  22  50  21  44  43  54  52  94  22  18   4  54  27

27  19  87  31  66  51  83  58 999  54  74  21  72  50  48  40  47  69  31  66  84  72
34  33  64  25  99  45  36  77  81  61  21  16  36  27  73  77   0   2  20  31  21  92
 9  81  65  32  71  79  11  32  36  14  90  92  59  90  92  97  62   2  91  21  34

82  10  66  45  74  17  64  10  62 999  94   0  82  93  79  45  38  14  36  23  62  22
85  46  51   2  80  87   0  40  87  89  69  91  80  94  27  79  49   1  25  12   2  94
 3  51  42  59  76  37  21  57  41  86  88  99  71  32  62  47  70  75  20  70   8

 5  70  29  63  16  12  56   1  28  37 999  58  40  25  75   2  85  68   3  87  47   0
98   3  97  75  90   5  88  32  15  48  41  68  93  87  85  47  91  22  86  18  41  31
89  61  34  45  13  11  97  25  75  30  15  21  11  18   1  88  62  73  91  87  33

82   8  82  89  73  37  70  36  78  77  26 999  70   6  74  13  80  67  75  50  81  11
76  34  10  30  47  85  92  68  41  30  29   5  96  20  58  35   3  18  50  88  90  11
63  49  87  38  50  82  26   3  48  88  17  65  31  72  66  48   6  20  20  70  47

47  84  78  74  44  44   0   2  89  78  21   8 999  44  62  67  95  66  54  55  54  39
77  88  52  61  81  96  56  41  88  11  80  10  21  23   7  84  13  24  79   7  21  90
 5  93  94  29  98  46  13  76  97  54  92  20  30  68  65   6  54  15   6   4  32
```

17 38 24 91 44 1 87 4 24 50 11 77 31 999 44 68 85 38 82 96 7 88
39 96 29 53 8 57 3 31 66 83 51 45 20 45 12 82 96 93 51 11 49 43
78 30 71 3 91 83 82 11 2 5 89 94 57 96 54 91 5 15 99 57 93

60 97 78 91 68 91 30 30 61 87 76 83 38 71 999 3 91 71 24 15 63 73
51 8 98 69 94 10 46 1 65 53 22 84 24 14 1 37 14 30 31 20 47 95
63 7 9 20 74 88 1 62 94 1 36 43 80 65 40 15 12 24 96 77 14

60 3 74 12 65 20 4 84 94 34 16 47 94 17 46 999 69 37 59 73 64 99
52 54 71 94 53 69 9 28 26 82 53 52 54 77 17 68 41 48 63 38 99 86
24 94 58 60 14 10 94 95 27 85 48 7 83 62 13 37 74 62 84 99 99

16 82 5 72 66 69 4 86 0 50 14 8 39 32 83 58 999 23 85 77 0 82
0 50 3 93 50 75 36 44 40 9 88 60 42 68 57 84 38 22 10 16 8 39
40 37 76 18 26 45 64 63 95 18 22 9 97 65 47 38 13 0 62 40 31

81 74 74 76 53 9 39 7 42 49 35 89 64 89 50 16 3 999 80 56 12 26
65 26 72 79 1 80 18 93 16 96 18 66 71 97 97 41 48 34 85 91 95 32
77 37 75 75 73 41 32 98 14 44 74 22 75 28 98 46 59 91 9 32 47

23 41 71 44 56 22 4 68 98 57 63 25 11 7 5 81 63 40 999 87 91 82
3 32 0 87 38 19 0 2 79 49 24 80 10 96 0 10 32 21 84 51 60 19
13 77 21 95 36 25 38 87 54 23 68 70 81 92 40 73 65 7 62 95 31

57 57 2 43 56 14 77 73 94 19 29 9 38 24 50 9 7 79 27 999 11 19
88 66 31 31 8 49 6 21 66 29 29 95 76 47 44 49 34 3 19 95 91 77
44 59 56 22 82 46 69 63 93 31 84 92 35 49 63 15 99 67 68 60 55

20 47 88 30 85 99 84 78 43 94 82 76 85 26 74 42 61 95 87 94 999 21
74 73 23 76 44 38 48 8 8 76 15 18 34 44 91 96 78 16 85 12 24 61
20 97 77 51 57 0 30 14 32 95 85 45 0 70 8 15 12 77 29 12 34

60 54 68 73 90 13 38 86 4 96 58 52 14 18 81 34 27 66 93 17 92 999
3 56 50 38 94 6 49 11 7 49 72 11 39 4 18 8 76 43 63 6 10 81
3 92 96 99 59 66 9 4 19 92 52 69 29 80 63 2 38 55 91 51 5

81 72 32 92 68 10 99 50 59 51 37 71 63 19 59 26 94 21 22 35 15 92
999 61 5 17 58 24 54 92 92 25 13 80 21 23 20 24 87 21 49 61 54 27
74 60 72 58 5 10 78 20 54 50 84 88 54 87 71 88 95 42 23 61 88

99 7 18 22 35 88 42 10 66 4 50 58 57 35 79 74 34 75 34 58 59 20
54 999 96 92 15 77 55 97 35 60 30 54 65 48 42 33 88 70 87 80 40 40
29 68 23 44 22 42 57 96 20 44 95 74 59 15 53 68 83 21 9 41 51

81 52 15 53 55 68 64 17 70 82 51 84 40 84 62 50 34 22 37 40 44 5
80 14 999 64 66 5 28 87 28 98 34 72 7 77 81 75 7 25 8 85 3 43
48 97 39 84 96 95 85 49 46 3 47 50 43 3 85 56 56 34 44 76 45

44 51 94 18 7 17 41 13 66 63 2 71 68 71 80 45 99 12 26 58 38 20
74 68 49 999 59 92 32 42 78 31 86 64 95 56 78 38 83 40 58 10 33 46
30 83 0 60 89 80 51 50 52 3 57 57 23 8 55 78 12 31 81 34 39

24 95 13 72 92 89 1 48 31 41 19 25 16 47 52 64 53 70 42 67 92 57
50 50 10 57 999 41 44 57 10 34 62 36 96 49 32 93 82 81 31 7 96 71
52 42 36 65 82 62 58 82 1 36 21 19 67 38 70 79 4 17 20 62 21

36 25 10 18 22 82 87 94 30 61 72 67 9 54 51 31 0 15 13 77 75 52
5 68 32 46 9 999 24 45 0 64 58 49 95 38 17 74 42 36 7 82 12 27
14 5 25 14 74 54 28 12 42 82 86 39 58 85 56 54 32 47 9 30 90

17 69 6 43 97 49 28 41 36 82 71 69 23 42 33 85 95 37 82 42 39 28
23 82 12 88 96 47 999 34 71 64 82 10 60 85 98 7 59 7 16 85 19 37
95 57 96 69 47 85 99 93 43 32 17 10 62 52 83 16 47 91 47 20 68

83 67 40 16 28 93 0 91 80 86 53 86 13 16 66 83 76 21 75 39 52 26
0 31 31 32 58 39 72 999 65 28 29 49 74 71 51 21 43 80 4 89 67 78
93 98 61 12 31 70 60 24 84 31 94 72 29 32 43 93 18 3 51 1 21

70 46 70 56 49 32 99 65 4 63 34 91 24 95 74 65 96 27 2 23 1 66
19 8 92 77 62 10 18 72 999 62 76 4 71 28 40 20 52 56 32 10 93 39
86 44 15 69 75 71 83 27 65 71 49 81 60 33 10 97 29 87 21 20 2

26 15 66 69 74 32 0 35 73 26 9 72 12 85 92 87 30 71 88 4 19 56
77 89 5 5 98 53 86 1 59 999 50 36 69 67 40 27 31 23 47 19 49 93
38 94 4 20 42 1 19 0 1 63 30 77 38 43 6 24 48 97 73 47 14

86 66 17 5 31 14 65 21 95 2 99 1 53 4 9 21 33 12 76 4 26 83
90 97 39 14 67 43 6 75 95 29 999 20 32 22 80 15 40 81 79 38 6 67
31 65 22 63 82 92 78 28 25 68 64 15 29 5 40 17 59 3 7 64 75

3 75 94 37 6 78 71 26 34 69 0 65 39 25 97 90 40 99 96 32 90 20
90 41 42 84 24 42 54 71 61 39 18 999 82 39 40 8 23 17 6 65 82 67
95 57 81 76 25 56 78 15 53 62 36 36 51 41 0 5 29 94 38 39 87

41 65 23 49 93 75 13 7 65 21 64 73 9 31 43 35 76 28 54 75 62 91
56 81 46 85 72 56 96 45 98 57 36 17 999 46 47 9 71 0 25 56 47 18
43 86 17 56 52 56 23 76 16 5 74 31 37 25 93 68 11 38 99 22 2

87 82 6 12 41 42 2 0 17 71 55 77 56 64 85 12 44 47 10 79 79 68
89 25 66 27 98 46 74 73 34 79 35 22 93 999 79 44 61 50 48 49 53 60
39 19 94 82 63 73 3 80 31 9 93 38 2 81 44 65 4 59 73 79 90

74 15 0 11 82 31 29 92 70 42 88 17 88 42 45 92 34 15 42 98 25 6
69 97 35 79 39 55 53 11 77 12 22 29 10 98 999 51 89 34 70 26 70 85
63 73 19 84 37 3 56 79 30 65 58 15 48 83 25 78 94 92 62 95 1

86 43 76 86 27 74 75 89 16 84 64 48 96 2 91 69 16 49 8 73 36 10
28 72 1 21 46 9 45 2 50 53 59 58 78 49 86 999 94 6 4 60 84 0
10 80 34 40 39 93 3 41 29 86 49 3 25 53 28 33 47 90 12 70 40

91 89 92 53 77 58 31 77 66 55 66 75 90 63 49 45 79 45 85 27 31 4
81 43 95 53 93 50 77 3 99 78 88 40 93 59 55 73 999 68 46 35 64 30

97 92 34 15 85 73 16 49 60 96 98 22 66 68 24 42 16 93 24 88 91

41 96 18 51 80 84 11 61 17 32 92 54 60 31 53 76 63 47 1 38 43 59
93 50 34 54 30 39 30 22 78 47 66 60 2 74 48 2 73 999 44 22 62 15
20 4 1 20 7 89 82 41 38 6 84 47 54 15 38 35 69 21 14 16 90

67 89 43 66 32 72 41 28 45 11 92 57 46 92 92 72 19 86 64 92 92 33
80 52 19 25 30 51 0 51 31 72 4 54 3 56 70 73 88 57 999 6 50 47
90 6 25 21 16 6 6 99 96 45 5 36 91 72 10 54 89 9 49 3 88

90 11 74 0 80 54 98 83 46 35 31 71 42 51 38 87 56 43 33 34 14 64
38 58 27 46 44 80 31 29 52 44 28 69 10 81 45 30 34 66 59 999 43 65
9 23 54 92 13 35 44 23 50 21 43 27 23 25 32 64 78 14 10 17 54

42 88 65 73 57 17 42 46 63 33 61 70 22 59 10 62 94 46 53 83 60 41
50 60 93 85 78 0 19 49 47 76 80 75 87 79 45 0 79 35 65 14 999 62
19 46 18 34 46 42 0 3 99 62 26 8 55 83 8 9 86 72 33 74 8

74 94 22 91 43 62 52 89 25 81 77 8 43 22 50 85 5 43 16 85 31 35
13 66 99 21 82 71 43 19 9 27 85 22 48 28 93 92 11 40 28 15 78 999
84 70 27 58 11 5 62 1 24 44 72 57 66 38 29 17 81 3 98 9 18

56 22 2 4 38 41 21 32 44 43 33 89 49 26 8 63 45 29 97 94 47 72
11 68 78 19 29 3 32 62 14 75 86 42 55 89 94 66 30 81 67 13 5 22
999 46 78 26 91 21 33 6 93 62 75 28 26 93 68 67 49 78 72 39 3

2 15 8 17 85 22 61 46 80 26 91 39 96 76 57 54 17 17 26 51 44 54
40 73 10 31 39 90 65 88 85 14 77 89 57 74 72 33 55 33 19 29 82 59
75 999 15 20 83 37 38 17 54 57 84 94 17 9 16 20 24 86 72 64 64

80 31 30 82 54 56 32 52 76 27 35 60 59 91 84 28 10 71 9 47 37 6
15 74 95 84 47 74 72 87 99 53 32 61 18 79 66 19 23 93 36 67 2 76
4 33 999 33 3 82 45 88 39 25 63 79 72 58 47 56 55 95 30 99 89

76 80 74 2 55 80 77 54 97 22 99 41 31 9 62 42 95 71 75 24 99 99
69 79 48 64 42 12 31 68 16 88 44 43 17 79 66 34 86 8 21 79 37 5
35 14 80 999 15 72 45 17 63 9 59 54 74 32 83 34 46 60 30 82 78

24 87 29 98 20 10 57 8 19 75 29 53 52 48 43 71 62 12 31 72 78 23
50 79 34 1 92 83 3 86 76 94 57 35 20 71 10 39 79 76 27 33 15 26
18 16 23 66 999 35 7 25 27 2 37 81 93 2 33 36 12 19 4 59 18

73 60 66 6 31 82 52 49 70 93 62 22 45 49 35 36 49 43 39 64 46 84
54 51 37 44 85 45 4 18 70 0 94 21 73 67 55 41 56 71 58 16 39 34
18 51 76 21 51 999 87 12 11 80 45 94 75 93 85 88 90 26 67 99 42

27 26 75 14 86 47 54 98 61 41 10 82 23 95 90 77 77 24 9 23 18 70
32 19 20 24 24 43 25 88 44 82 0 68 72 95 72 69 19 26 66 29 70 10
86 85 0 96 51 89 999 61 82 39 79 85 89 81 46 2 96 50 88 85 6

33 26 94 89 98 75 6 95 53 27 8 90 6 60 11 18 86 3 97 27 99 55

81 59 32 23 62 33 15 4 20 68 67 61 99 98 86 67 53 53 54 28 80 86
88 28 43 55 80 69 9 999 57 55 87 28 50 86 41 8 98 46 14 87 41

29 20 12 52 66 16 38 59 58 47 37 85 6 62 23 31 27 43 85 30 5 93
49 26 42 7 3 27 2 58 6 2 3 68 52 73 70 59 93 47 76 12 28 73
74 41 32 45 70 98 8 32 999 87 32 13 28 51 10 48 16 52 79 83 37

29 84 68 7 4 38 59 28 4 76 65 57 79 6 94 39 9 57 27 41 8 57
36 66 27 24 35 84 66 89 64 43 67 36 76 64 45 25 92 21 66 70 56 37
77 4 48 31 1 97 90 59 80 999 63 13 69 33 9 87 81 54 33 78 92

1 41 52 74 42 95 96 56 2 79 40 8 0 11 47 83 52 41 4 84 55 71
21 69 90 40 93 45 67 90 0 68 4 35 71 98 18 88 55 71 44 70 86 24
80 44 55 29 0 69 42 2 26 43 999 84 10 4 78 37 46 25 41 87 95

99 36 2 89 97 85 96 47 13 64 61 95 94 83 51 51 96 43 40 98 19 7
37 75 9 1 98 6 52 69 7 72 38 83 80 89 11 61 63 77 16 32 27 62
17 75 65 35 16 40 98 44 36 22 65 999 98 24 8 90 78 48 55 43 45

89 93 39 71 97 29 50 91 49 27 78 58 69 72 30 82 60 90 79 8 84 52
32 46 53 8 86 7 50 50 61 33 45 44 5 31 97 36 23 66 32 24 70 69
70 59 5 32 13 76 76 50 71 94 81 35 999 96 4 93 95 65 38 67 64

20 56 70 57 25 94 35 22 37 11 57 16 16 19 88 2 18 31 80 22 50 47
38 88 70 9 45 75 72 72 85 9 84 4 24 1 32 88 17 19 61 33 89 39
72 40 92 50 70 70 76 36 7 76 4 71 82 999 25 75 99 94 7 20 18

28 48 85 34 78 34 14 29 23 45 46 42 45 75 45 81 24 62 52 64 4 48
2 26 92 36 78 19 84 64 56 13 64 65 33 94 24 47 50 32 59 28 74 53
75 48 86 38 85 90 20 1 72 64 26 39 84 45 999 80 75 71 30 57 28

96 15 86 72 4 47 65 84 33 73 79 92 78 46 72 43 82 94 23 73 47 59
57 36 66 36 71 99 60 30 66 37 13 8 50 55 92 87 74 17 93 61 34 75
42 93 35 27 68 59 38 95 42 5 19 54 62 18 3 999 55 54 55 76 61

26 11 98 57 67 54 46 7 12 55 38 6 91 50 90 62 82 93 69 42 99 71
51 41 25 42 61 59 33 97 91 92 71 69 1 49 45 68 25 37 84 3 24 85
58 40 66 31 52 15 83 16 76 64 38 54 58 53 57 58 999 77 49 62 70

36 48 92 26 82 54 2 69 23 70 56 35 78 92 71 60 62 69 81 43 28 89
48 87 53 70 35 90 79 39 54 40 97 32 35 81 71 36 86 38 0 40 90 78
91 33 60 16 16 80 59 12 19 21 92 17 26 6 70 86 29 999 61 47 92

91 52 58 99 47 69 29 23 76 11 81 88 77 91 70 42 43 55 72 36 85 69
31 28 13 84 32 1 29 46 33 22 36 28 84 18 77 81 62 81 29 6 18 71
89 99 81 28 58 22 13 17 99 8 37 80 7 84 86 21 14 23 999 51 8

10 97 34 11 7 65 34 9 65 90 48 53 81 21 57 68 65 94 66 24 71 83
29 8 54 99 35 20 29 27 86 1 12 78 59 70 85 48 5 13 7 21 85 92
51 20 73 67 56 67 82 13 23 73 44 58 92 76 69 99 52 87 37 999 25

```
62  76   9  83  97  27  82  78   3  35  70  34  98  32  54  73  47  20  74  36  74  54
94  85  77  48  49  55  72  73  85  52   1  14  94  30   1  97  50  65  53  55  14  88
94  22  67  47  74  17  18  33  20  43   6  53  55  26  62   6  46  85  58  93 999
```

A solution to the 65 city problem is:

```
1-60-5-38-14-6-25-58-36-57-59-23-49-63-42-4-8-48-44-52-18
 -27-53-13-7-55-31-21-50-32-30-64-24-10-12-61-35-40-19-37-65
 -9-39-22-45-28-17-62-41-29-56-26-47-43-51-33-20-3-34-11-16
 -2-15-54-46-1
 at cost 136
```

C.5 Problem n100a

This is the cost matrix for problem n100a:

```
999 23   8  24  40  29  27  43  13  17  97  96  61  44  36  41  89  57  42  73   8  76
 38 14  69  77  55  72  78  96  73  35  52  81  27  14  66  52  67  25  37  13   4  64
 50 69  36  51  66  61  81  33  30   7  49  56   9   3  13  57  90   3  47   3  83  13
 80 74   7  73  35  77  40  54   6  49  91  94  46  99  98  55  45  19  68  81  16  75
 11 85  62   2  27  11  15  50  94  64  93  12

 83 999 80  99   1  43  83  48  73  61  26  21  95  38  93  98  47  31  31  83  27  63
 91 98  24  44  26  24  21  57  61   3  88  32  25  87  35  72  93  12  50  23  73   7
 50  5  73  52  90  85  64  26  37  91  71  61  62  45  20  52  94  75  74  26  51  77
 59 15  26  87  56  59  79  26   5  74  12  56  39  54  39  78  42   8  35  54  51  34
 12 55  31  65  91  20  57  14  72  55  27  98

 23 79  999 87  29  84   1  56  56   6  78  53   2  78  41  94  23  91  13   8  43  22
 13 53  93   3  65  42  36  11  29   9   6  80  76  55  34  41  96  23   7  90  63  85
 62 11  90  79  22  80  62  29  23   0  12  18  54  58  85  75  62  84  36  76  38  59
  7 36  30  19  48  65  13  66  68  23  16  62  26  86  48  46  97  15  49  71  27  84
 61 60  21  82  13  91  38   1  49  58  35  89

 81 36  72  999  8  80  69  53  33  50  15  36  35  39  78  49  86  10  86  19  73  55
 82 84   3  12  29  79  84  25  15  25  55  31  38  41  38  13  43  76  61  98  71  21
 74 89  94  14  33  67  81  99   5  98  34  82  60  65   9  85  88   4   6  40  18  96
 56  6   7  72  36  36  35  98  89  36  82  56  60  93   1  94  84  31  30  51  38  96
 51 15  67   7  10  91  45  13  46  36  10  72

 64 77  68  22  999 61  35  49  17  72  18  43  17  14   4   7  60  94  94  13   5  57
 86 27  71  74  79  86  79  18  55  77  62  53  44   0  36  93  70  19  57  48  84  79
 23 40  90  55  54  74   7  55  54  73   5  69  61  68  22  19   6  59  96  19  17  82
  9 59  74   0   1  23  96  13  22  49  78  30  50  86  91  77  58  97  95  72  82  61
  9 79   1  10  63  44  17  58  28  58   8  49

 67 61  41  74  89  999 36  24  67  19  81  29  14  73  40   7   9  38  42  32  60  32
```

```
20  20  27  85  27  48  33  35  14  29  42  15   5  46  39  48  81  74  23  49  69  11
74  52  31  78  69   5  19  14  45  98  38  11   8  81  42  95  40  67  86  92  54  87
40  12  74  66   7  20   8   8  85  92  59  63  46  82  60  80  82  61  48  26  11  34
 2  86   1   8  68  67  34  72   2  83  22  42

91  54  39  98  54   1 999   2  75  11  59  48  29   2  46  96  22  25  20  40   5  58
41  34  80  19  32  85   7   9   5  18  90  22  40   1   4  65  36  68  10   0  43  26
82  20  91   0  79  89   1   0  68  67  89  15  48  91  73  95  92  35  59   9  79  56
 5  41  62  18  87  50  84  48  39  72  68  44  36  52  97   2  40  89  90  57   2  64
86  35  23  18  82  16  17  27  77   4  81  42

56  60  65  57  33  69  52 999  33  73  80   5  85  48  90  62  36  69  60  61  51  51
55  29   4  58  56  82  79  59  97  15  49  57   6  11  17  60  95  41  94  71  67  84
18  49  92  74  37  60  49   6  58  71  79  92  66  61  31  31  22  15  89  75  37  71
86  39  33  23  80  79  67  35  85   0  45  22   8   1  79  42  66  51  78  54  60  26
15  57  94   3  92  71  24  78  79  89  28  61

58  44   2  93  59  94  86  10 999  43  36  59  86  19  57  29  43  22  80  64  70  93
39  64   6  72   1  32  91  11  98  66  24  24  29  27  60  11   2  95  76  68  98  51
89  41   3  25  27  79  41  89  42   2  28  33  53  59  51  99   3  33  44  74  40  78
56  77  75  27  76   6  22  92  98  50  91  93  74  78  53  17  73  14  10  78  30  33
87  29  47   0  49  40  33  62  94  78  82  27

18  70  24  53  79  30  63  43  19 999  18  20  89  37  19  84  44  99  86  86  69  21
73  87  99  85  25  62  84  69  97  72  42  18   5  10  43   7  78  97  34  22  93  86
92  62  51  25   5  12  54  92  47  34  28  16  48  52  12  23  28  14  90  32   8  90
60  98  20  39  50  47  55  63  53  45  29  48  19  30  43  73  74  72  17  74  78  50
19  38  88  33  52  84  55  47  67  67  23   4

 1   6  36  71  59  59  71  23  15  12 999  15   1  28  59  33   6  62  90  23  53  77
18   2  34  26  96  73   9  50  32  49   8  40  73   5  36  49  45  43  77  66  42  76
52  86  94  52  69  74  98  70  61   5  52  38  27  84   5   4  54  41  86  55  14  31
 7  44  81  20  70  33  23  43  92  86  41  28  52  47  63  50  48  48  22  75  17   3
76  40  67  23  26  16  48  88  85   5   9  51

27  26   4  32  69  72  53  41   2  91  32 999  30  83  56  59  92  92  15   8   6  33
41  71  83  49   4  87  62  79  34  82  36  35  39   2  65  15  55  46  88  41  29  44
76  13  66  85  56   3  77   1  87  14  40  41  20  50  87  26  15  79  27  62  59   7
49  66  15  20  31  29  16  79  24  84  25  32  59  78  37  62  98  98  77  22  95  67
13  71   7   1  51  92  30  22  25  56  42  52

36   6  80  30  97  45  75  32  83  14  51  26 999  50  55  36   8  45  95  72  12  59
95  59  37  44  56  65  97  54   6  89  69   6  84  50  72  82  73  38  21  80  21  98
54  85  77  77  33  51  20   6  65  98  45  59  41  20  61  87  64  78  21  14   4   1
26  97  51   4  44  57  35  49  10  61  50  15  27  56  88  87  48  37  24  21  91  80
70  43  96  96  94  76  77   1  69  31  27  14

96  75  82  33  73  49  34  71  41  28  23  34  42 999   8  61  13  85  61  43  80  35
 7  42  24  32  41  22  54  89  98  81  68   2  87  51  53  87  38  67  67  83  87  15
57  46  35  57  18  43  34  55  77  97  66   1  80  85  81  52  30  76   2  34  63  34
11  27  60  46  38  10  92   5   9   5  95  37  15   2  77  30   4  21  42  99  40  74
```

```
42  98  15  67  75  57  77  32   7  23  68  25

52  37  97  59  27  34  86  63   7  34  30  82  17  11 999  46  82  67  37  47  85  60
82  77  47  95  44  49  73  49  70  57  75  53  33   0  34  25  67  26  42  73  10  22
14  38  21  17  87  32  60  97  18  93  17  12  80  45  70  43  86   7  74  91  12   1
21  91  15  79  31  48  34  93  20  63  39  80  25  52  90  10  63   3  81  65  16  19
97  74  82  82  27  75   3  30  48  62  10  78

10  20  58  53  76  59  72  27  55  87  94  68  28  92  99 999  36  78  13  62   0   4
80  13  83  72  16  94  14  70  22  23  32  15  48  49  56  10  84  59  44  55  44  97
97   9  21  31  70   5  71  36  80  51  21  78  30  98  98  82  30  12  55  36  90  90
71  68  18  57  35  20  46  27   1   5  83  85  71  12  34  82  53  89  58  96  76  81
35  67  61   8  79  44  93  42  52  87  69  50

47  48  99  84  95  47  88  22  13  52  98  77  31  88  38   5 999  84   9  45   7  73
72  86  75  41  19  18  52  46  45  95  95  99  57  59  62  13  87  36  89  28  14  80
46  51  33  82  37  51   0  84  59  17  60  55  24  11  62  71  95  31   6  36   8  41
99  44  51  57  11  82  46  12  19  19  24  55  48  10   5  70  55  69  26  71  13  94
62  30   7  49  18  32  29  55  92  90  66  11

94  10   7   0  99  28  22  38  31  63  24  52  96  95  39   1  12 999  66  98  62  33
54  99  35  18   0  37  87  52  73  58  69  59  21  26  40  95  89  57  49  42  94  15
46  66  63  56  67   7  35  43  60  78  82  26   1  10  20  44   1  27  36  18  85  27
 2  42  21  75  87  31  84  13  29  95  31   4  22  54  11  53  38   3  58  77  63  84
75  26  72  56  42  57  97  36  79  97  29  90

31  17  25  63  99  53  48  37  23   1  35  61  29  42  62  60  70  40 999  61  13  74
19   0  42  95  51  28  70  59  54  73  97   2  69  56  15  71  21  26  38  38  50  79
73  77  17  32  20  75  14  83  61  69  89  11  27  11  16  47  62  58  33  71  17   2
42  48  21  67  61  36  84  69  10  10  70  96  70  64  17  29  12  86  52  51  22  10
29  86  33  48  58  95  55  54  28  72  84  45

62   6   2  14  92  42  73  31   4  17  82   5  41  56  16  38  22   7  89 999  64  25
48   9  76  83  77  55  88   3  23  12   6  95  67  56  59   2  27   4  59  35  92  49
68  14  56  78  91  47  70  43   7  55  70  25  86  70   2  22  30   7  71  41   4  82
79  54  37  82  95  31  94   3  61  49  93  78  98  45  72  79  65  58  96  15  26  92
51  61  22  63  71  62  40  24  26  64  27   8

69  24  90  85  76  87  56  17  88   2  57  37  58  81  77  26  94  93  30  27 999  55
 9  43  34  65  53  90  44   6  93  24  87  18  48  97  94  32  13  94  66  84  86  13
39  40  91   5  18  87   5   1  45  61  10  63  44  61  24  22  90  89  61  97  38  34
66  70  94  31  11  13  66  82  50  86  39  56  91   9  94  97  27  63  26  15  32  96
84  75  75  39  81  38   4  66  66  70  82  35

19  92  59  43  33  45  10  64  91  88  71   1  99   1  58  81  91   7  91  84  67 999
51  70  17   5  59  34  71  86   5  59  59  34  48   7   3   2  98  42  11  18   0  29
49  93  50  28  85  23   9  75  22  52  28  61  40  19   5  39   9  70  85  23  78  77
75  17  29  61  97  78  28  41  38  42  78  54  18  88  85  75  74  84  59  52  10  96
60  83  80  15  30  71  16  21   1  52   1  91

50  83  30  78  22  54  92  31  66  12  46  49  24   2  34  57  65  30   3  15  69  83
```

```
999 63  69  40  97  38  96  42  94  61  10  80  19  12  30  37  81  34  86  47  15  69
53  54  25  33  16  70  34  69  50  32  82  15  83  70  67  53  50  23  75  12  62  93
14  24  78  21  49  82  48  76  94  88  94  63  99  98  20  21  97  69  92  52  73  48
52   0  25  91  41  99   4  50  47  48  55  10

93  67  18  34  61  66  34  35  66  12  16  34  36  22  34  30  95   1  18  46  39  39
60 999  48  26  34  20  54  73  25  87   9  41  66  85   2  49  15  34  34  15   0  65
97  14  33  46  74  34  95  45  48  58   9  51  86  18  64  99  53  56  97  33  58  76
81  77  12  44  45  78  12  54  30  42  82  40  35  21  86  44  51  85  66  95  64  22
23  19  30  19  19  88  86  73  61  84  22  95

38  54  99   3  54  15  80  69   6  12  62  57  85  90  59  34  34  80   7  54  47  30
74  53 999  39  41  74  36  13   7  61  34  29  63  41  96  47  30  98  45  61  72  76
12  13  29  13  61  90  51  62  75  44  75  18  15  46  27  64  65  44  78  80  94  88
31  95  30   5  15  38  95  54   1  56  22  83  71  16  29  27  89  14  32  56  70  49
52  20  98  80  61  11  26  65  44  93  93  33

24  38  54  90  53   7  27  34  46  67  61  72  75  12  86  61  89  28  27  95   9  25
85   2  35 999  69   5  42  84  37  31  88  54  90  19  85  47  85  86  39  20  68  64
26  66  99  42   1  63  67  46  89  38  61  98  31  24   4  28  41  17  95  32  79  45
67  74  27  76  39  54  45  73  55   6  60   7  82  33   9  21  79  91  69  49  94  83
90  75  65  67  97  47  53  91  46   5  35  11

88  72  27  19  90  12  24  40  96  33  37  79  57  12  93  82  45  96  82   2  90  56
95  54  51  11 999  12  85  53  61  94  82  48  72  26  13  61  98   2  10  93  16  25
 8   1   7  12  98   0  31  77  21  65  88  76  81  81  48  52  63  52  89  42  63  60
44  23   8  38  42  14  88  51  65  14  56  21  58  26  17  22  37  75  18   7  75  42
32  73  99  68  51  16  86  37  89  10  54  88

52  86  73  16  69  38  52  56  39  18   2  96  34  68  34  84  47  90  68  97  83  82
71  33  82   7  99 999  36  26  64  76  35  37  27  38  24  79  67  51  82  13  15  63
10  76  68  79  91  37  48  43  42  70   4  72  71   5  11  56  58  60  22  14  60  98
71  33  21  33  80  73   6   9  24  79  51  61  96  60   1  60  68  20  41  56  71  15
49   2  31  42  20   5  18   7  73  13  85  75

52  71  99  27  81   5  77   9  36  99  23  31  94   0  94   9  86  95  56   8  48   6
96  89  70   3  33  28 999  28  33  21  32  24  96  83  59  18  83  30  58  77  95  29
19  69  81  58  67  37   2   1  29  64  89  88  48  52  19  84  19  29  21  55  18  78
34  92  81  31  73  28  89  49  46  34  13  99  35  44  72  77  16  56  61  41  67  81
51   5  14  90  23  94  83  20  72  50  39  31

40  89   9  70  11  25  38  95  75  44   6  65  52  26  71  76  81  98  96  66  55   2
 6  43  82  85  33  34  43 999  72  57  71  16   3  73   8  86  41  20  38  33  39  14
47  50  34  25  53  39  40  80  91  27  10  72  43   8  10  96  39   5  39  58  23  26
66  51  86  26  96  36  69  14   1  55  80  62  43  31  52  87  82  26  28  24  27  84
27  14  31   4  66  78  80  43  73  85  18  44

36  62  56  82  55  78  94  64   3  69  62  66   7   4  89  56  69  17  15  70  69  79
14  34  99  69  51  37  76  46 999  79  24  11  74  10  22  12  94  62  91  13  38  94
68  96  10   1  65  95  63  66  90  83  83  12   4  79  66  71  86  61  60  48  44  85
63  85  72  53  89  96  27  92  49  44  98  98  26  46  56  41   5   2  56  88  43  56
```

43 0 14 57 35 58 80 74 94 24 44 67

12 68 32 1 35 50 4 44 48 65 4 23 79 99 99 73 44 8 75 23 87 16
4 25 52 41 61 80 43 11 85 999 73 14 83 22 60 93 82 20 22 98 21 98
85 56 49 90 31 21 8 61 5 52 98 45 50 1 17 23 12 38 22 40 51 6
97 60 6 83 36 67 37 56 63 67 64 57 59 5 22 49 86 56 42 26 6 0
9 34 70 91 88 33 86 42 12 47 14 81

51 70 84 64 46 90 90 96 67 52 86 25 20 47 16 77 49 40 92 14 38 99
20 21 40 91 86 49 65 19 42 89 999 19 55 90 16 68 60 41 79 55 98 8
9 34 26 45 47 95 72 61 50 72 68 92 3 22 20 61 64 45 96 98 21 17
52 35 46 21 85 31 76 0 19 27 42 31 92 21 51 56 37 40 56 45 8 17
0 24 68 77 44 58 48 65 84 30 50 90

59 51 11 73 10 17 55 98 87 48 43 0 47 59 31 9 97 14 77 30 69 54
94 14 65 34 69 3 51 90 13 47 14 999 66 9 24 45 75 19 66 60 13 37
43 59 77 89 40 25 10 94 73 8 38 32 10 81 33 45 8 61 76 17 90 20
44 10 77 42 50 33 29 62 54 88 16 73 30 57 93 39 51 66 94 77 42 60
95 38 85 90 48 91 17 78 87 36 [illegible] 4

99 96 1 61 45 45 94 41 89 28 49 57 48 23 26 67 7 39 15 32 47 96
76 41 63 84 85 56 99 38 28 41 26 40 999 1 13 53 11 22 64 41 70 83
54 42 81 83 91 6 94 20 42 76 66 27 98 34 49 31 55 5 23 66 31 34
37 70 40 89 15 21 18 17 67 92 89 66 3 33 27 18 39 34 31 37 76 53
85 44 65 92 87 41 70 30 40 39 87 40

34 37 94 9 57 39 12 47 44 0 90 89 24 10 9 45 79 15 43 35 86 81
12 80 3 63 76 44 97 26 91 58 67 73 70 999 28 62 97 4 5 80 80 0
22 20 21 37 25 79 76 63 17 77 41 3 90 5 87 56 63 91 54 17 17 14
30 51 54 37 58 71 34 97 62 72 38 84 96 26 21 61 14 50 70 94 25 53
96 32 46 43 73 64 87 0 56 33 59 28

93 49 54 46 46 75 27 67 3 76 17 45 98 38 76 75 0 65 67 11 51 31
30 20 55 93 25 12 65 54 36 9 79 24 61 65 999 36 34 74 93 45 80 6
65 67 62 30 51 40 78 81 28 26 64 85 56 56 74 76 90 74 51 89 55 12
45 91 91 45 38 59 8 62 8 89 58 86 93 0 84 89 81 14 20 99 29 10
70 18 99 89 5 67 6 2 78 22 88 97

28 96 59 94 26 44 32 65 97 25 3 67 58 73 13 54 76 35 7 61 30 14
52 10 66 69 81 20 36 35 24 33 13 90 81 79 75 999 12 16 71 22 0 39
97 12 85 8 93 80 74 51 51 93 80 76 65 80 25 42 34 48 8 15 29 81
11 28 3 34 49 80 81 [illegible] 45 91 80 34 45 24 96 87 39 23 42 30 83 41
49 55 52 66 95 39 24 85 52 5 19 99

9 4 72 76 77 31 70 26 95 97 72 79 30 85 99 96 92 66 20 42 95 68
21 9 78 27 4 42 75 24 57 8 60 95 61 11 78 67 999 68 83 42 92 15
83 7 99 2 18 8 72 51 35 73 31 46 35 76 80 28 66 72 10 14 73 14
46 56 87 76 97 42 16 42 28 86 42 3 84 79 81 58 76 99 99 61 86 55
80 70 45 72 88 29 8 62 56 79 71 28

98 99 65 13 51 73 96 60 50 6 1 39 85 27 8 53 76 41 84 26 96 69

58 43 69 65 89 80 63 17 6 82 30 15 18 13 94 1 9 999 10 96 4 41
6 6 89 33 97 64 3 2 13 39 96 40 46 91 40 34 69 44 18 4 49 29
1 36 53 87 13 26 72 96 62 32 15 64 10 57 10 57 71 22 53 45 67 26
48 35 86 86 69 85 89 36 95 1 32 36

26 87 85 27 14 90 11 3 19 55 9 91 73 5 75 96 61 22 54 73 83 37
40 90 40 23 58 22 42 76 92 1 8 95 81 9 63 46 27 18 999 46 82 94
48 94 53 21 39 35 57 13 52 77 61 8 93 17 67 45 71 50 79 5 16 15
81 90 88 47 24 18 35 75 27 25 91 9 5 47 35 16 80 29 19 54 95 40
76 17 67 90 39 5 94 31 64 82 30 18

94 56 76 69 93 95 13 91 73 89 25 3 73 95 62 1 83 40 23 55 54 7
52 58 64 57 98 3 2 90 70 63 14 12 14 34 85 46 39 95 25 999 31 2
40 37 10 50 89 4 85 53 9 43 96 94 96 45 86 57 60 13 99 69 44 77
64 42 7 82 12 15 0 99 98 50 83 49 78 2 5 57 37 85 99 28 44 73
80 19 38 68 41 21 48 71 70 73 60 24

87 49 24 91 24 64 83 10 58 0 63 16 85 70 78 46 59 60 55 45 2 53
19 51 1 83 26 32 99 38 11 83 10 24 95 96 39 8 10 17 3 43 999 8
69 18 56 82 22 95 92 80 73 46 88 2 53 89 23 69 47 21 59 26 37 52
75 49 35 71 33 31 65 89 72 6 81 51 10 22 27 54 11 20 3 85 58 54
16 98 95 88 58 71 72 8 92 62 41 57

50 38 30 29 5 28 79 61 45 16 13 37 99 84 6 50 18 77 7 40 3 93
98 80 48 13 38 27 86 59 94 8 34 75 72 17 85 68 23 24 31 37 32 999
24 68 9 0 68 77 40 70 56 6 9 21 53 40 31 60 49 9 67 39 97 51
78 49 39 40 34 64 28 52 58 47 83 92 22 81 76 18 77 95 25 44 54 94
30 4 61 67 66 19 82 19 42 41 95 37

2 36 84 99 17 32 11 27 6 53 21 42 26 51 13 51 41 95 8 13 44 83
0 98 40 58 40 74 33 56 94 95 79 63 52 91 51 71 53 23 52 96 80 43
999 56 2 30 21 18 93 56 41 47 96 94 18 93 35 49 82 42 88 36 36 34
39 9 69 69 41 59 72 74 21 6 74 8 65 98 78 40 36 20 42 66 40 49
34 78 46 54 52 59 9 99 18 35 70 95

32 23 94 61 61 55 57 84 34 79 18 8 5 88 46 30 49 11 56 93 10 10
51 38 30 82 27 89 92 41 26 27 80 79 21 79 27 81 90 28 48 72 7 72
10 999 6 36 53 61 71 22 4 27 70 98 57 95 92 65 61 83 76 1 91 46
51 64 90 93 94 69 36 82 44 93 94 50 98 15 61 47 85 31 19 84 2 31
7 28 16 89 28 10 98 48 77 34 0 30

56 0 34 98 34 32 63 44 30 29 35 17 98 42 67 83 99 94 77 54 53 8
63 69 60 38 57 15 52 43 30 42 99 39 11 77 60 97 40 40 93 74 18 99
49 15 999 32 54 55 92 69 27 1 88 53 59 11 17 70 74 30 45 96 66 26
50 91 29 22 9 41 33 18 93 79 49 33 37 6 64 91 31 76 38 93 83 54
13 28 15 49 57 74 45 99 8 33 62 6

8 98 5 3 23 24 15 9 11 10 80 30 73 95 42 30 35 80 10 81 69 11
51 69 56 40 14 18 50 4 18 54 89 6 16 17 46 62 15 2 89 12 17 59
53 84 82 999 92 57 14 56 63 37 48 61 3 19 68 61 51 86 48 15 93 75
36 15 17 86 25 89 43 47 43 55 45 84 88 54 66 33 87 73 7 71 51 2

54 26 45 57 36 49 60 64 80 96 50 5

52 34 6 86 19 8 12 35 15 89 33 46 36 68 32 13 39 33 85 77 24 77
99 14 62 19 56 98 34 6 15 23 18 60 14 2 12 54 65 68 18 2 30 31
57 1 84 43 999 57 96 73 11 7 36 62 73 96 84 76 64 40 55 4 76 90
39 48 20 99 96 56 63 46 98 11 0 66 47 83 6 77 76 55 11 31 90 42
14 55 5 80 62 78 13 91 5 25 90 60

72 55 97 76 99 34 60 33 30 75 76 45 66 33 29 99 2 54 21 16 79 67
56 52 77 92 50 62 48 59 11 62 11 21 23 21 21 3 18 6 44 59 50 54
12 31 91 89 27 999 62 35 31 83 75 18 65 9 89 24 4 52 7 72 51 36
33 57 92 76 85 38 28 55 91 26 98 6 27 70 38 14 7 98 75 7 0 82
10 77 6 20 9 91 34 23 14 68 55 46

63 12 28 37 22 22 75 84 10 40 27 54 5 40 3 4 8 65 46 18 35 53
78 82 9 9 84 18 22 78 18 95 80 78 87 36 96 93 11 0 89 77 81 2
78 4 96 2 94 73 999 61 1 12 57 39 50 40 87 12 94 9 54 15 37 26
89 39 97 14 97 58 74 7 13 50 88 65 51 59 73 70 50 76 51 34 31 47
84 11 36 92 91 27 69 2 26 57 66 67

65 65 17 80 98 8 77 81 59 62 12 63 51 11 44 68 63 47 28 94 76 37
72 69 98 14 47 66 15 34 24 8 88 56 50 67 60 44 36 31 70 54 72 13
1 29 12 24 43 21 58 999 6 42 2 76 14 5 68 95 24 80 55 20 6 58
83 47 37 36 68 87 36 91 79 30 41 79 44 38 86 37 79 13 97 54 98 32
1 53 3 73 24 89 94 37 57 47 66 77

43 66 7 65 84 92 67 70 7 33 80 76 67 71 37 35 19 48 53 22 55 53
29 58 77 70 58 67 0 31 13 23 33 46 61 39 70 63 69 29 38 3 34 17
34 75 28 87 18 51 4 33 999 15 31 73 77 24 79 31 31 24 66 52 90 82
99 66 8 7 62 57 27 91 93 4 50 29 95 10 14 68 49 22 50 67 18 52
42 1 28 65 20 29 48 68 65 35 43 30

44 86 26 92 67 18 27 32 33 77 81 93 79 78 89 19 96 9 54 68 86 80
57 53 27 62 85 68 20 59 53 70 84 49 4 20 73 2 25 61 26 18 26 80
27 51 6 79 56 85 59 51 21 999 88 66 25 75 20 56 12 12 60 60 37 67
46 82 51 18 24 98 76 20 36 70 59 60 52 1 90 88 25 92 16 46 57 46
19 61 43 23 52 80 68 29 90 34 41 61

40 25 28 20 31 43 94 93 42 94 13 31 81 48 63 23 99 77 45 15 74 12
54 20 92 17 11 13 92 24 52 13 71 70 85 54 88 92 72 15 94 60 94 78
50 27 55 98 60 51 6 83 71 89 999 23 25 93 66 51 90 44 14 33 90 49
14 4 55 10 94 64 4 37 46 86 51 35 85 35 7 48 15 29 88 60 23 95
88 70 19 6 33 19 79 28 52 28 1 49

77 37 50 40 0 81 53 10 87 79 15 81 12 97 10 94 92 99 46 37 43 95
57 33 74 66 64 55 67 26 90 51 58 77 87 21 87 25 68 49 80 62 12 53
22 1 87 48 17 82 55 55 60 60 8 999 47 38 32 11 48 14 78 29 7 35
10 41 6 93 21 64 21 83 41 5 61 54 71 87 68 77 79 85 74 20 81 19
93 7 9 3 56 83 40 83 73 69 93 45

40 85 14 28 68 76 71 89 23 98 43 28 94 13 26 37 29 47 27 86 83 26

57 86 70 37 60 31 40 14 13 35 19 45 66 69 66 67 83 28 40 59 42 57
61 25 1 46 6 9 88 62 57 94 8 40 999 99 15 71 42 58 83 37 46 15
28 31 42 89 20 15 91 59 81 78 60 38 41 21 81 98 82 14 10 18 77 35
82 31 67 32 63 9 48 32 46 47 61 16

59 3 26 39 74 77 52 89 49 57 78 75 81 52 86 72 83 67 94 49 63 34
19 86 98 73 41 39 76 52 90 85 45 8 71 1 80 34 0 94 49 9 81 54
82 58 81 54 16 67 93 30 99 23 73 67 32 999 59 86 71 7 42 93 23 11
2 66 86 9 18 1 5 7 87 89 90 93 85 39 5 29 15 13 66 43 27 39
67 68 7 3 45 80 44 31 68 54 43 11

19 98 32 35 91 84 11 97 55 85 21 40 59 59 35 88 49 51 85 70 24 27
53 0 84 2 67 41 86 17 20 69 59 50 57 61 38 59 0 97 58 3 61 76
74 30 47 77 44 74 22 31 44 17 96 12 84 81 999 99 65 24 69 52 98 74
37 89 81 56 64 3 37 10 86 50 42 93 13 70 63 54 0 74 60 10 24 33
37 9 62 75 24 90 21 60 76 5 33 3

38 12 48 64 24 66 15 48 16 85 32 21 23 86 22 89 38 34 58 60 94 0
88 76 50 76 73 20 44 58 43 24 83 26 78 27 28 17 65 59 17 4 96 44
98 55 63 30 25 80 11 32 66 99 4 66 79 35 78 999 80 71 19 77 20 50
74 71 27 76 92 49 2 36 59 64 46 14 78 48 6 53 68 58 2 59 56 26
52 63 69 35 43 79 7 42 6 73 21 94

5 87 92 82 9 46 19 48 36 60 12 36 9 98 43 95 58 39 58 96 24 58
40 33 50 57 50 83 15 70 85 36 56 75 26 76 94 92 3 41 50 52 86 87
75 16 63 99 11 32 66 32 83 0 95 40 68 13 12 43 999 55 95 76 0 46
42 5 16 47 28 19 23 20 99 35 66 19 30 86 34 74 56 42 18 48 96 6
78 95 87 29 35 17 17 89 7 65 57 77

49 77 55 36 25 9 31 31 33 73 18 51 78 55 56 23 37 13 68 30 53 93
75 46 6 38 44 44 64 73 8 34 96 3 46 36 27 78 45 18 50 15 23 55
34 26 37 18 5 23 79 49 21 18 33 6 37 70 96 82 59 999 56 80 36 40
59 83 97 62 82 56 12 20 67 68 48 79 22 93 74 43 34 64 65 25 96 77
97 43 77 1 29 37 96 15 16 34 23 27

82 13 42 14 70 72 70 23 36 51 8 13 43 39 49 41 37 89 77 47 43 72
70 63 39 21 41 82 76 33 91 14 75 49 78 54 51 41 39 26 50 66 47 81
82 79 85 64 77 90 9 79 72 77 6 62 83 24 37 9 98 25 999 37 88 52
21 2 40 87 60 8 99 6 27 33 5 53 85 41 35 22 23 42 32 21 9 24
54 57 37 90 52 25 67 46 4 9 65 25

64 14 99 55 0 8 54 51 17 58 17 95 28 26 33 14 60 29 21 77 71 9
77 5 55 82 82 21 28 95 70 56 5 76 62 36 34 84 84 60 44 36 62 37
96 3 15 2 42 40 0 29 57 80 3 49 93 91 63 45 63 53 8 999 98 9
54 51 92 14 19 8 6 8 99 49 3 67 20 86 73 81 70 67 60 59 61 31
23 32 50 1 77 61 75 33 89 74 66 41

43 62 16 63 70 52 95 1 83 78 65 9 96 11 11 13 61 75 36 26 87 31
30 57 25 77 45 56 59 30 6 38 21 43 35 76 64 48 49 82 71 75 18 69
35 54 88 26 73 72 41 39 1 5 3 26 79 17 35 98 36 63 1 14 999 37
65 56 88 91 27 19 85 48 15 44 96 82 27 67 31 48 97 36 71 52 0 17

```
37  41  87  74   6  82  86  56  45  52  80  51

26  60  79  71  41   5  62  29  45  96  75  86  56  53  72   1  73  93  86  56  85  60
45   9  37  83  71  41   8   6  12  45  98   3  54  76  47  83  91  87  18  30  96  56
59   1  27   9  14  75  25  45  38  31  49  12   5  28  83  38  15  86  64  85  69 999
84  94  10   2  66  88  81  35  35  19  89   3  52   3  47  49  43  39  11  25  31   8
 7  38  96  80  62  41  26  50   7  50  98   1

22  70  54  12  32  69  61   0  42  13  14  84  31  83  73  78  54  81  16  71  46  83
 4  79  37  34   3  57  35   6  70  36  30  20  26  99  14  39  86  51  69  23   6  31
29  71  41  80  14  75  90  92  62   4  28  56  91  75  16  49  81  52  55  19  60  88
999 11  60  61  21  82  12  72  29  97  10  98  21  14  46   3  38  68  40  81  18   8
15  56  11   3  65  82  53   6  31  43  41  12

53  55  40  59  47  50  30  34  66  84  58  61  36   2  11  15  23  85  24  47  15   5
91  53  23  92  60  71  10  79  15  54  72  74   4  51  84  11  58  78  55  21  57  78
34  34  29   6  29  41  75  46  46  81  66  33  81  93  73  30  89  76  51   1  95  13
72 999  54   2  36  82  37   5  32  47  61  88  22  64  35  86  42  70  68  23  64  53
25  41  48  13  90  92  47  52   1  68  25  56

 2  51   7  69  67  24  61   3  33  34  37  57   4  30  30  65  10  50  78  39  58  13
49  16  60  34  43  27  49  85  48  53  85  30  50  56   3  27   8  90  82  68  44  79
97  76  43  97  96  94  79  76  39  18   5  36   7  29  19  15  86  48  12   6  60   2
24  14 999  26  72  30  19  26  14  20  75  48  14  35  54  14  81   0  85  26  52  82
80  33  12   6  92   0  34  68  15  14  57  65

80  99  62  34  54  15  87  46  52  99  11  20  74  15  61  43  90  52  88  70  69  68
11   1   3  58  31  88  61  96  73  78  93  32   5  87  48  58  97   9  55  42  38  30
52  48  48  82  45  69   8  76  14   4  13  77  23  78  35  32  44  22  60   5  93  79
20  12  97 999  66  11  33  27  83  85  87  93  25   8  32  77   4   1   0  36  67  68
11  65  85  86  85  15  68  84  96  57  44  41

14  19  45  56  46  17   8  42   1  91  89  74  78   7   2   3  95  81  43  15  86  77
58   0  17  74  41   2  74  61  45  34  26  17  28  86  85  68  82  83  38  22  45  66
42  65  19  58   3  77  48  53  22  69  51  33  22  74  46   2  63  12  76  84   1  76
18  79  88  96 999  84  50  20  49  97   1  54  43  90  49  22  66  68  43  11  27   1
21  47  54  50  96  63  22  82  48  82  34  41

 9  69  18   7   0  90  85  77  17  49   4   0   6  35  47  77   2  17  31  55   8  33
67  19  84  42  71  86  24  97  47  94   6  43  81  91  66  45  24  48  51  63  76  81
36  54  66  84  40  13   0  46  77  34  52  47  34  77  73  25  57  98  86  97  27  34
 6  56  54   3  85 999  71  95  44  24  41  83  71  87  28  62   6  50  82  26  68  34
26  66  47  73  19  31  25  43   5  56  70  76

 1  81  10  30   1   1  40  11  39  38   7  62  78  32   8  34  87  94   1  20  46  96
72  21  63  74  43  91   3  69  16  80  39  39  93  43  95  12  40   6  80   3   1  88
98  38  71   8  64  60  95  34  57   0  76   1   3  58  35  57   3   2  43   1  34   0
72  60   1  38  71  72 999  47  16  80  53  20  28  87  90  98  82  94  89  43  10  55
27  28  63  73  86  48  51  63  31  89  40  55

98  35   6  97  14  73  99  30  92  67  78  53  38  21  26  32  44  50  22  78  90  81
```

77 17 69 83 68 88 59 93 47 71 29 40 40 90 31 70 10 23 89 96 56 99
76 40 69 34 12 20 87 16 3 76 68 87 55 76 1 6 8 81 53 2 40 25
87 90 95 62 22 85 85 999 27 54 3 59 16 28 13 88 94 87 49 25 14 44
52 39 31 90 78 71 3 29 58 67 94 61

82 43 88 79 34 67 43 24 72 80 53 65 68 89 39 56 93 73 7 20 6 69
11 35 66 39 49 93 66 3 82 92 48 14 69 6 5 76 25 46 27 34 2 17
55 10 92 96 36 73 46 26 67 78 45 91 72 41 32 68 42 87 22 10 82 81
69 65 94 35 49 91 86 13 999 28 32 86 3 49 61 74 49 40 97 2 82 92
48 39 36 71 57 59 12 10 1 47 51 67

65 13 89 22 57 79 46 11 30 26 33 38 36 72 56 73 80 72 79 73 45 79
89 35 38 75 85 69 10 12 33 56 58 2 61 49 34 68 64 10 11 98 56 14
98 69 22 12 28 91 60 44 53 77 95 8 61 44 3 24 45 15 70 12 35 18
63 98 72 27 64 63 51 77 2 999 34 13 5 40 70 42 97 22 17 69 55 18
85 59 65 9 27 71 70 6 86 34 7 56

64 45 32 9 55 98 29 30 13 10 64 37 7 28 13 23 75 50 11 59 8 37
34 31 93 10 60 86 79 66 75 55 12 91 29 67 21 47 42 41 69 77 31 10
98 67 19 67 91 6 12 49 34 41 31 86 71 47 9 90 71 87 9 89 98 68
72 39 7 36 90 93 12 26 32 54 999 97 56 83 9 9 82 94 25 34 64 61
70 37 26 77 97 90 24 26 73 78 65 19

25 38 76 71 26 50 65 65 33 69 61 97 57 96 11 38 86 22 6 47 97 31
47 64 23 43 55 56 40 71 62 68 28 90 86 71 31 89 26 65 48 65 27 74
36 53 83 90 5 31 75 36 70 56 42 43 49 19 27 93 73 58 63 86 52 14
68 32 16 31 65 72 63 43 42 81 22 999 12 19 8 17 37 59 67 6 13 62
89 94 17 99 48 41 18 68 50 34 79 92

18 42 8 1 67 81 80 70 5 73 0 31 83 94 95 18 25 39 33 57 50 6
54 72 85 95 28 69 34 81 32 76 11 21 86 12 32 34 63 83 61 71 74 23
47 9 14 24 27 25 77 66 82 74 58 59 96 39 28 82 28 66 67 46 52 11
19 33 61 95 30 28 78 8 22 90 72 37 999 74 57 6 99 5 20 77 18 5
46 25 62 86 83 51 28 26 71 6 30 68

27 45 31 27 35 81 99 63 58 99 19 42 40 53 17 69 81 32 89 14 62 37
35 69 44 99 79 29 27 20 6 84 48 52 82 82 35 47 9 79 74 74 98 63
7 44 52 56 99 49 79 58 79 87 83 42 24 72 60 46 71 2 54 84 78 22
89 41 39 7 84 79 82 51 83 61 65 47 95 999 47 84 6 9 85 82 51 77
97 81 75 50 91 29 13 43 84 59 32 56

90 1 81 27 42 13 89 53 7 65 50 55 54 6 46 38 22 25 61 45 25 94
18 94 93 10 35 20 24 21 97 26 33 15 13 88 6 65 49 89 22 37 98 23
20 30 22 91 82 77 37 96 47 2 51 4 27 62 6 89 15 93 4 7 56 55
0 67 72 88 33 34 38 47 22 12 51 77 24 27 999 21 13 88 58 47 75 61
94 82 8 21 35 26 91 16 44 85 67 29

52 8 96 83 4 67 45 77 49 48 61 50 6 74 55 77 42 40 35 8 14 30
18 92 29 25 96 94 12 64 77 39 49 41 95 73 16 68 85 74 78 14 91 31
57 26 30 24 76 4 35 89 12 51 35 59 92 11 42 43 60 19 83 64 75 0
14 25 85 14 64 74 32 75 67 7 44 41 10 37 77 999 12 88 6 39 8 11

14 63 45 94 72 99 52 62 20 72 8 14

27 64 24 90 34 91 28 49 22 29 95 74 44 8 71 28 37 79 46 73 70 98
18 8 28 78 48 66 81 31 51 9 37 36 67 15 27 53 60 57 28 11 47 48
98 86 83 20 93 85 20 31 98 74 88 99 95 44 38 23 77 25 47 76 24 22
55 31 84 4 60 67 78 55 33 63 37 78 81 52 26 45 999 35 95 54 83 5
77 33 81 99 5 58 82 8 51 25 41 16

6 57 44 12 18 67 96 7 62 59 0 24 8 29 45 7 91 18 32 86 9 2
74 91 24 30 44 48 62 51 17 57 20 86 92 64 76 59 99 99 62 79 89 42
60 96 43 36 17 26 85 42 55 7 38 18 36 88 6 5 29 82 7 31 93 36
17 59 28 58 70 22 60 69 33 89 44 61 27 97 51 3 19 999 18 66 51 30
31 9 17 81 52 0 39 98 42 57 21 74

6 25 72 45 32 90 26 2 17 52 31 56 15 43 23 97 18 24 13 1 63 82
51 31 24 83 48 19 1 16 51 67 77 9 86 58 36 33 28 19 80 71 18 55
30 54 92 67 81 99 12 8 26 41 88 72 18 73 39 25 5 53 53 21 71 14
14 52 87 45 17 21 19 47 70 51 57 56 29 63 72 89 1 38 999 79 45 75
67 30 32 15 81 30 62 43 43 57 70 18

65 85 58 84 70 49 83 43 4 97 29 47 98 23 52 9 83 63 65 48 68 20
43 16 78 95 77 94 90 95 59 61 37 73 12 77 6 77 10 66 75 62 61 76
74 62 37 1 0 62 15 35 76 7 49 90 81 37 8 32 5 44 98 44 12 80
59 1 88 70 54 70 63 76 84 42 42 76 89 48 16 75 13 17 42 999 80 24
21 85 72 41 7 17 93 92 73 31 78 91

14 60 32 29 91 97 46 74 84 39 22 6 94 56 19 93 28 35 54 58 34 54
96 74 90 69 30 71 94 6 54 44 62 19 61 45 73 76 57 15 70 84 52 17
51 84 18 29 30 10 43 44 88 89 52 77 73 57 70 28 26 48 9 30 84 77
51 4 64 32 76 67 52 50 45 78 44 58 47 2 66 71 21 63 19 89 999 61
99 73 62 26 79 57 69 23 5 0 96 59

31 17 66 16 65 74 92 14 69 59 44 59 8 40 43 96 21 31 23 94 32 56
15 42 0 47 18 23 32 26 27 8 51 91 46 37 23 65 51 34 94 98 74 22
48 9 98 6 19 4 90 68 77 84 33 58 27 32 42 51 17 63 35 83 74 14
75 79 28 80 50 11 21 28 11 84 32 83 26 49 49 19 95 79 40 68 47 999
77 56 81 90 60 78 34 2 0 73 90 89

61 11 91 65 47 6 89 87 40 81 0 6 57 31 30 9 7 10 12 10 68 74
73 45 53 73 50 32 39 80 59 61 52 14 30 57 7 1 62 7 63 27 47 63
92 9 99 36 39 46 72 57 94 63 88 92 76 68 45 76 7 87 34 94 0 51
13 69 54 49 97 2 8 41 36 76 76 55 57 17 48 22 28 41 17 94 14 18
999 94 45 43 70 45 10 65 97 64 15 49

98 13 20 52 77 12 41 68 6 65 65 41 59 50 17 49 65 26 2 43 7 79
22 18 71 88 61 10 77 53 11 32 23 44 76 54 35 91 50 61 67 45 63 25
35 12 87 90 53 80 45 8 16 22 28 40 82 38 7 28 42 55 13 48 41 30
75 3 81 5 36 97 91 62 97 59 24 43 99 63 78 93 73 75 33 24 3 28
28 999 57 46 88 37 37 10 0 67 56 27

39 22 97 10 71 93 29 42 94 67 71 95 0 95 50 0 22 80 69 95 58 16

5 97 92 37 16 98 53 1 70 42 39 62 24 58 71 37 21 68 54 43 31 58
25 49 63 60 60 15 84 96 62 71 8 75 54 23 20 49 85 28 48 86 86 46
54 42 73 44 12 33 35 33 13 83 15 63 39 82 42 98 82 38 60 36 84 22
21 75 999 24 47 70 48 97 16 99 82 38

36 10 81 36 59 73 80 10 8 1 50 20 59 58 20 51 1 29 39 86 36 81
71 85 16 35 47 90 88 72 13 73 86 32 1 25 24 9 68 41 11 39 37 71
69 38 42 94 1 31 25 61 43 26 61 71 24 96 64 59 11 5 15 3 49 17
75 44 34 54 78 60 81 69 36 31 53 13 27 28 34 52 89 86 61 88 61 66
78 53 84 999 71 23 32 36 16 94 92 23

85 19 24 61 44 82 59 44 58 47 59 35 53 40 67 80 5 67 4 54 65 68
2 50 19 53 31 22 51 95 77 54 72 4 43 34 76 91 94 44 81 43 52 99
8 30 73 29 21 33 81 92 13 21 56 15 62 25 13 86 25 37 81 56 24 73
3 16 26 22 48 1 23 93 8 50 22 30 20 85 4 24 98 96 65 76 18 57
90 55 98 46 999 69 16 62 72 26 3 41

60 72 36 87 30 5 9 52 53 31 38 82 15 51 58 78 50 59 47 12 32 21
14 10 50 28 4 42 94 26 28 24 53 74 1 62 36 79 62 39 75 87 64 86
71 61 52 74 76 13 24 18 18 4 8 13 80 26 45 24 99 99 9 64 51 41
97 82 66 57 50 5 85 85 58 84 54 24 82 51 32 55 9 4 10 86 96 40
91 3 31 4 14 999 54 2 9 75 61 43

73 60 50 64 86 18 42 82 48 52 29 47 42 6 67 8 95 85 8 61 65 68
42 26 98 51 63 18 79 52 34 5 21 39 35 51 22 42 46 6 38 36 32 41
54 21 3 74 1 74 78 53 66 66 44 5 79 43 94 67 66 71 75 22 0 7
48 21 66 98 36 11 39 62 80 51 31 6 3 35 24 58 99 42 94 70 84 54
6 2 78 80 35 91 999 77 10 4 87 24

8 32 89 33 17 86 35 54 21 57 52 85 96 19 67 4 26 38 6 60 0 27
27 44 49 89 9 52 60 17 87 96 42 14 56 68 24 39 14 66 90 21 87 79
35 31 21 57 27 56 30 74 78 14 34 83 57 68 16 52 42 9 32 60 73 75
6 23 40 14 7 87 88 49 72 34 9 26 83 86 31 18 65 14 34 27 60 52
50 84 77 48 98 78 65 999 7 82 52 48

88 49 15 7 90 74 7 38 5 50 26 73 46 80 91 71 46 90 78 52 3 62
58 87 20 82 34 78 94 10 15 25 3 43 67 26 75 27 98 82 33 66 74 28
62 7 87 77 16 80 72 51 52 59 76 78 66 24 87 84 73 19 46 13 58 26
49 91 73 60 34 35 38 40 2 67 82 76 54 74 85 6 0 76 46 27 4 83
60 29 92 47 27 34 59 25 999 83 42 65

82 45 74 28 97 51 2 30 95 21 39 45 27 60 89 42 63 94 47 41 1 95
46 34 78 69 39 81 41 95 47 39 96 51 95 66 69 22 19 66 35 92 12 50
31 94 53 66 5 36 64 59 33 71 97 47 77 43 78 69 72 17 36 51 48 79
44 6 19 73 2 43 27 2 99 3 50 23 2 97 42 75 40 71 82 39 51 57
25 57 19 92 2 94 19 2 38 999 12 1

47 82 80 77 80 28 93 0 71 17 26 12 13 29 80 36 55 17 96 16 11 89
46 58 55 80 46 42 14 16 35 6 89 94 96 71 17 50 63 39 62 32 33 71
48 78 47 72 37 41 85 80 45 54 79 83 1 21 41 35 1 83 91 23 73 96
21 59 50 16 13 0 89 94 17 11 93 82 3 92 8 26 0 3 15 55 51 73

```
 9  77  26  96  62  57  25  44  45  91  999 71

58  99  38   6  94  34  39   1  36  24  52  73   0  68  81   7  57   2  12  60  70  38
 1  98  71  53  77  77  21  91  88  73  93   5  45  67  91  91   4  19  30   9  16  45
60  85  77  26  21   4  62  96  72  77  81  64  78  19  64  15  94  98  26   0  66  27
84  20  54  19  92  20  49  69  24  38  96  36  23  23  21  15  29  11   5  91  68  14
50  90  21  38   3  94  81   8  22  84  20  999
```

A solution to the 100 city problem is:

```
1-58-39-78-19-34-28-55-99-61-54-80-62-92-10-100-13-96-21-52-45

 -23-95-65-63-68-70-84-82-66-16-75-86-49-77-69-94-35-3-7-42

 -73-6-91-30-22-37-17-51-15-36-44-48-40-38-43-41-32-88-25-31

 -90-97-83-93-72-12-9-27-50-87-98-79-4-81-67-8-76-59-26-24

 -18-57-47-2-5-71-60-85-20-74-53-29-14-56-46-64-33-89-11-1


 at cost 132
```

Bibliography

1. Abdelrahman, Tarek S. and Trevor N. Mudge. " Parallel Branch and Bound Algorithms on Hypercube Multiprocessors," *The Third Conference on Hypercube Concurrent Computers and Applications*, 32: 1492-1499 The Association for Computing Machinery, 1988.
2. Aho, Alfred B. and others. *The Design and Analysis of Computer Algorithms* Reading MA: Addison-Wesley Publishing Company, 1974.
3. Antonoff, Michael. "Software by Natural Selection," *Popular Science*: 70-74 (October 1991).
4. Beard, Ralph A. *Determination of Algorithm Parallelism in NP Complete Problems for Distributed Architectures.* MS thesis, AFIT/GE-90D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.
5. Bell, Gordon. "The Future of High Performance Computers in Science and Engineering," *Communications of the ACM*, 32:1091-1101 (December 1989)
6. Bomans, Luc and Dirk Roose. "Benchmarking the iPSC/2 hypercube multicomputer", *Concurrency* :3-18 (September 1989)
7. Bourgeois, L. and J. Lassalle. "An Extension of the Munkres Algorithm for the Assignment Problem to Rectangular Matrices", *Communications of the ACM*, 14 (December 1971)
8. Brassard, Gilles and Paul Brantley. *Algorithmics: Theory and Practice.* Englewood Cliffs NJ: Prentice Hall, 1988
9. Carpenter, Capt Barry A. *Implementation and Performance Analysis of Parallel Assignment Algorithms on a Hypercube Computer.* MS thesis, AFIT/GE-87D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.
10. Christofides, Nicos *Graph Theory: An Algorithmic Approach.* London: Academic Press, 1975
11. Cvetanovic, Z. and C. Nofsinger. " Parallel Astar Search on Message-Passing Architectures," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, 1 , Architecture Tract,*: 82-90 (January 1990)
12. DeCegama, Angel L. *Parallel Processing Architectures and VLSI Hardware Volume 1.* Englewood Cliffs NJ: Prentice Hall, 1989.
13. Felten, Edward W. " Best-First Branch-and-Bound on a Hypercube," *The Third Conference on Hypercube Concurrent Computers and Applications*, 1: 1500-1504, The Association for Computing Machinery, 1988.
14. Garey M. R. and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completness.* W. H. Freeman and Company, San Francisco, 1979.
15. Hays, John P. and Trevor Mudge. "Hypercube Supercomputers," *Proceedings of the IEEE Vol. 77, No. 12:* 1829-1840 (December 1989).
16. Hennessy, John L. and Norman P. Jouppi. " Computer Technology and Architecture: An Evolving Interaction," *IEEE Computer, 2*: 18-29, 1991
17. Intel Corporation. Parallel Programming Primer. Order number 311914-001. Beaverton Or, March 1990.
18. Jansen, J. M. and F. W. Sijstermans. "Parallel Branch-and Bound Algorithms," *Future Generation Computer Systems*, 4: 271-279 (December 1989).

19. Korf, Richard E. "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, 27: 97-109 (1985)

20. Kumar,Vipin, K. Ramesh and V. Nageshware Rao. " Parallel Best-first Search of State-Space Graphs: A summary of Results," *Automated Reasoning*: 122-128

21. Li, Guo-Jie and Benjamin W. Wah. "Computational Efficiency of Parallel Combinatorial OR-Tree Searches," *IEEE Transactions on Software Engineering Vol. 16 No. 1*: 13-30 (January 1990).

22. Ma, Richard P. and others, "A Dynamic Load Balancer for a Parallel Branch and Bound Algorithm," *The Third Conference on Hypercube Concurrent Computers and Applications, 2*: 1505-1513, The Association for Computing Machinery, 1988.

23. Miller, D. L. and J. F. Penky. "Results From a Parallel Branch and Bound Algorithm for the Asymmetric Traveling Salesman Problem," *Operations Research Letters, 8*: 129-135 (July 1989).

24. Mraz, Capt Richard T. " Performance Evaluation of Parallel Branch and Bound Search with the Intel iPSC/2 Hypercube Supercomputer. " MS thesis, AFIT/GE-90D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

25. Pangas, Roy P. and Wooster E. Daniels. "Branch-and-Bound Algorithms on a Hypercube," *The Third Conference on Hypercube Concurrent Computers and Applications, 2*: 1505-1513, The Association for Computing Machinery, 1988.

26. Pennington, R. J. and others. "Parallel Implementations of a Branch and Bound Algorithm for the Optimization of Distributed Database Computer Networks" *IEEE Region 5 Conference*. Piscataway, NJ: IEEE Service Center 1988.

27. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison Wesley Publishing Company, 1985.

28. Quinn, Michael J. "Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer," *IEEE Transaction on Computers, 31*: 384-387 (March 1990).

29. Ragsdale, Susann. and others. *Parallel Programming Primer*, Intel Corporation, order number 311914-001.

30. Saletore, Vikram, A. *Machine Independent Parallel Execution of Speculative Computations*. PhD. dissertation. University of Illinois at Urbana-Champaign, Urbana IL, 1991.

31. Schwan, Karsten and others. "Process and Workload Migration for a Parallel Branch and Bound Algorithm on a Hypercube Multicomputer," *The Third Conference on Hypercube Concurrent Computers and Applications, 2*: 1520-1530, The Association for Computing Machinery, 1988.

32. Stone, Harold S. and John Cocke. "Computer Architecture in the 1990s," *IEEE Computer, 2*: 30-38, 1991.

33. Work, 2LT Paul R. *Parallelizing Serial Code in a Distributed Processing Environment with an Application in High Frequency Electromagnetic Scattering*. MS thesis, AFIT/GCS-91D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

VITA

Capt Joel S. Garman was born on January 25, 1954 in Glasgow, KY. He joined the Air Force on July 18, 1974 and was assigned to the 90th Strategic Missile Wing at F.E. Warren AFB, Chyenne, WY as a Minuteman III missile systems analyst. He was accepted into the Airman's Education and Commissioning Program (AECP) and graduated from the University of Central Florida in May of 1985 with a BSEE degree. After Officer's Training School (OTS), Capt Garmon was assigned to SA-ALC as an electrical engineer working on the F-5, T-38, T-37, OA-37, OV-10, and O-2 aircraft systems.

Permanent address
9434 Valley Way
San Antonio, TX 78250

March 1992 Master's Thesis

Implementation and Analysis of NP-Complete Algorithms on a Distributed Memory Computer

Joel S Garmon, Captain, USAF

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GE/ENG/92M-01

Approved for public release; distribution unlimited

The purpose of this research is to explore methods used to parallelize NP-complete problems and the degree of improvement that can be realized using different methods of load balancing.
A serial and four parallel A* branch and bound algorithms were implemented and executed on an Intel iPSC/2 hypercube computer. One parallel algorithm used a global, or centralized, list to store unfinished work and the other three parallel algorithms used a distributed list to store unfinished work locally on each processor.
The three distributed list algorithms are: without load balancing, with load balancing, and with load balancing and work distribution. The difference between load balancing and work distribution is load balancing only occurs when a processor becomes idle and work distribution attempts to emulate the global list of unfinished work by sharing work throughout the algorithm, not just at the end. Factors which effect when and how often to load balance are also investigated.
Which algorithm performed best depended on how many processors were used to solve the problem. For a small number of processors, 16 or less, the centralized list algorithm easily outperformed all others. However, after 16 processors, the overhead of all processors trying to communicate and request work from the same centralized list began to outweigh any benefits of having a global list. Now the distributed list algorithms began to perform best. When using 32 processors, the distributed list with load balancing and work distribution out performed the other algorithms.

Search, Hypercube, Parallel, NP-complete 188

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL